Cortex-M3 权威指南

Joseph Yiu 著

宋岩 译

译序

我接触 ARM 的历史约 4 年，早期是很欣赏这类处理器，到了后来切身使用它们的机会越来越多，慢慢地有了感觉，也更加喜欢了。在偶然听说 Cortex-M3 后，我就冥冥地感到它不寻常。只是因为其它工作一直没有去了解它，直到今年初才进一步学习，很快就觉得相知恨晚。当时只能看 ARM 官方的重量级资料，在看到这本书的英文原稿后，更感觉被电到了一样，于是突然有了把它翻译成中文的冲动。经过累计约 150 个小时的奋战，终于有了此译稿。在翻译过程中，我始终采用下列指导思想：

1. 尽量使用短句，并且把句子口语化。我认为高深的道理不一定要用高级的语法句型才能表达。想想看，即使是几位博士互相聊天讨论一个课题，也还是使用口语吧，而且火花往往就是在这种讨论中产生呢！
2. 多用修辞方法，并且常常引用表现力强的词汇——甚至包括网络用语和脍炙人口的歌词。另外，有时会加工句子，使得风格像是对话。这样做的目的是整个文风更鲜活——有点像为写出高分作文而努力的样子。这点可能与很多学术著作的“严肃、平实”文风不同，也是一次大胆的尝试。还希望读者不吝给予反馈。
3. 在“宏观”上直译，在“微观”上意译。英语不仅单一句子的语法和汉语不同，并且句子的连贯方式也与汉语不同。因此在十几个到几十个单词的范围内，我先把它们翻译成脑子里的“中间语言”，再把中间语言翻译成汉语。这样，就最大地避免了贻笑大方的“英式汉语”。
4. 有些术语名词不方便翻译成汉语，或者目前的翻译方式不统一，或者与其它术语翻译的结果很接近，如 error 和 fault，就只能用英语意会。此时我就保留英文单词，相信这样比硬生生地翻译成汉语还好。这些词汇主要是:retarget, fault, region 等。另外，英文中有一个很能精练表达“两者都”意思的单词及其用法：”both”，我也常常保留之。
5. 图表对颜色的使用比较丰满，尤其是比较大型的插图，相信这样能帮助读者分析和理解。插图是从原图直接复制的，因此矢量图变成了位图，无法再适应任何比例的缩放。不过，我在复制原图时，把原图以 200%的比例放大，从而提高了图片的质量。
6. 我在很多地方加了译注。比较短的译注就直接以“()”加在文字后面。比较长的译注则为它开出一个“文字池”，放到相应的“.text”后面并与之相临。早期的译注多用于解释一些不是很广为人知的术语，后期的译注则更多是我认为有必要补充的内容，包括读者在阅读过程中可能会产生的问题，容易混淆的概念，深入理解等。
7. 我对少量自然段作了改编。也有个别部分译自 ARM 提供的权威文档。

本书的翻译工作在 40%进度的时候是最困难的时期，有一种好像长跑中遇到了所谓“极限”的感觉。望着距离掉下去还有那么高的滚动条，甚至都有停住的自我暗示了。那天刚好是哀悼日的第一天，我本来情绪很低沉，但在我看到默哀完毕，天安门广场上排山倒海般地呐喊“中国加油”时，我突然有了强烈的共振感觉，那是一种热泪盈眶的激动和感叹，甚至觉得他们就是在鼓励我！让我一下子振作起来，找回了比刚开始还要强烈的干劲，并且更加信心满满。这种精神力量一直推动我翻译完最后一个字，并且还有“余勇可贾”的快感^_^

整个翻译的时间跨度是在 2008.05.10-2008.06.07，共计 28 天。不知这是否算得上很“仓促”。想必有很多句式还能改得更好，甚至还有错别字等低级错误。我使用了五笔输入法，可能错别字会错得很离奇，不过肯定逃不过读者雪亮的双眼的。希望读者在发现错误后批评指正。反馈地址是：rock.song@hotmail.com，也可以通过QQ:9471202/9312500。

本译稿草稿完成后，我交给几位好友去试读和审校，得以揪出了很多大大小小的 bugs。他们是：浮云，土豆波，美眉 Y 和小胖，在这里以点名表示感谢！

宋 岩 2008.07.02

原作序

谁是最节能，最擅长把好钢用在刀刃上的人？要让我说，我一定得表一表单片机的开发者。他们使出浑身解术，写出精妙玲珑的代码，把单片机点点滴滴的力量汇集起来，让它如同涌泉一般尽情地迸发，灌溉滋养着各行各业。是什么灵丹妙药赐予了他们这么神奇力量？除了好的处理器之外，还要配合好的开发环境和工具链。也正出于此，在设计ARM7TDMI处理器时，ARM的工具链工程师们和CPU设计师们强强联手，为了让它的内部结构更优化、更精练、更到位而并肩奋战了很多日日夜夜，终于有了ARM7TDMI的无限辉煌，并且久经岁月的洗礼依旧光芒绽放。

珠联璧合的最新果实，是破茧而出的ARM Cortex-M3处理器。这个小尤物，处处闪耀着ARM体系结构激动人心的新突破。它基于最新最好的32位ARMv7架构——这个架构支持高度成功的Thumb-2指令集，还有很多时尚、前卫甚至崭新的特性，充满了新生代的气息。它在很好、很强大的同时，编程模型却变得更加清新爽洁了。不管你是祖国的花朵、是人民教师、还是精明的商人，也无所谓是新手还是骨灰级玩家，Cortex-M3都将尽情展现它的秀外慧中，带给你喜出望外的收获和“激活”！

ARM嵌入式解决方案主任

Wayne Lyons

前 言

不管你是做软件的还是做硬件的，只要相中了ARM的Cortex-M3处理器，这本书就是为你而写。以前Cortex-M3的资料只有两个大部头，分别是：

- 《Cortex-M3技术参考手册》（Cortex-M3 Technical Reference Manual，简称Cortex-M3 TRM）
- 《ARMv7-M应用程序级架构参考手册》（ARMv7-M Application Level Architecture Reference Manual）

虽然这它俩差不多是权威到"古文观止"级的，但实在是太深入了，以致于对新手来说那简直就是天书。本书则是一个精简版，并且叙述的前后更有条理。目标读者包括：一线程序员，嵌入式产品设计师，片上系统（SoC）工程师，嵌入式系统发烧友，学院研究员，还包括所有涉猎过单片机和微处理器领域，慧眼识珍看中了Cortex-M3的人民大众们。

本书要给Cortex-M3的架构做一个简介，浏览一下指令系统，写几个段代码练练手，说一些硬件特性，再表一表该处理器精深的调试系统。本书还给出了应用程序范例，手把手地教你使用开发工具，包括ARM的工具和GNU的工具链。如果你以前是ARM7TDMI的玩家，正准备着升级装备到Cortex-M3，本书也非常解渴，里面讲述了两者的不同，以及鸟枪换炮的升级过程。

缩略语

缩写代号	含义
ADK	AMBA设计套件
AHB	先进高性能总线
AHB-AP	AHB访问端口
AMBA	先进单片机总线架构
APB	先进外设总线
ARM ARM	ARM架构参考手册
ASIC	行业领域专用集成电路
ATB	先进跟踪总线
BE8	字节不变式大端模式
CPI	每条指令的周期数
CPU	中央处理单元
DAP	调试访问端口
DSP	数字信号处理器 / 数字信号处理
DWT	数据观察点及跟踪
ETM	嵌入式跟踪宏单元
FPB	闪存地址重载及断点
FSR	Fault状态寄存器
HTM	CoreSight AHB跟踪宏单元
ICE	在线仿真器
IDE	集成开发环境
IRQ	中断请求（通常是指外部中断的请求）
ISA	指令系统架构
ISR	中断服务例程
ITM	指令跟踪宏单元
JTAG	连结点测试行动组（一个关于测试和调试接口的标准）
JTAG-DP	JTAG调试端口
LR	连接寄存器
LSB	最低有效位
LSU	加载/存储单元
MCU	微控制器单元（俗称单片机）
MMU	存储器管理单元
MPU	存储器保护单元
MSB	最高有效位
MSP	主堆栈指针
NMI	不可屏蔽中断
NVIC	嵌套向量中断控制器
OS	操作系统
PC	程序计数器
PSP	进程堆栈指针
PPB	私有外设总线

本书大面积地使用了如下的排版字体约定：

- 普通汇编代码

```
MOV R0, R1      ; 把寄存器R1中的数据移至R0
```

- 以模式化语法表示的汇编代码——编程时必须使用真实的寄存器名字

```
MRS <reg>, <special_reg> ;
```

- C 程序代码

```
for (i=0;i<3;i++) { func1(); }
```

- 伪代码

```
if (a > b) { ...
```

- 数值:

1. 4'hC , 0x123 都表示16进制数
2. *#3*表示数字3 (e.g., IRQ #3 就是指3号中断)
3. *#immed_12*表示一个12位的立即数
4. 寄存器位。通常是表示一个位段的数值，例如
   bit[15:12] 表示位序号从15往下数到12，这一段的数值。

- 寄存器访问类型
  1. R 表示只读
  2. W表示只写
  3. RW 表示可读可写（前3条好像地球人都知道）
  4. R/Wc 表示可读，但是写访问将使之清 0

其它参考资料

1． *Cortex-M3 Technical Reference Manual (TRM) (Cortex-M3技术参考手册)*
请从www.arm.com/documentation/ARMProcessor_Cores/index.html下载

2． *ARMv7-M Architecture Application Level Reference Manual(ARMv7-M应用级架构参考手册)*
请从www.arm.com/products/CPUs/ARM_Cortex-M3_v7.html下载

3．*CoreSight Technology System Design Guide（CoreSight技术系统设计指导）*
请从www.arm.com/documentation/Trace_Debug/index.html下载

4． *AMBA Specification (AMBA规格书)*
请从www.arm.com/products/solutions/AMBA_Spec.html下载

5． *AAPCS Procedure Call Standard for the ARM Architecture(AAPCS ARM架构过程调用标准)*
请从www.arm.com/pdfs/aapcs.pdf下载

6． *RVCT 3.0 Compiler and Library Guide(RVCT 3.0编译器及库向导)*
请从www.arm.com/pdfs/DUI0205G_rvct_compiler_and_libraries_guide.pdf下载

7． *ARM Application Note 179: Cortex-M3 Embedded Software Development(ARM应用笔记179：Cortex-M3嵌入式软件开发)*
请从www.arm.com/documentation/Application_Notes/index.html下载

占位符 1，为目录保留

占位符 2，为目录保留

占位符 3，为目录保留

占位符 4，为目录保留

占位符 5，为目录保留

占位符 6，为目录保留

占位符 7，为目录保留

占位符 8，为目录保留

第1章

介绍

- ARM Cortex-M3处理器初探
- ARM的各种架构版本
- 指令集的开发
- Thumb-2指令集架构(ISA)
- Cortex-M3的舞台
- 本书组织
- 深入研究用的读物

ARM Cortex-M3 处理器初探

单片机市场的规模可以用“巨无霸”来形容，预计到2010时每年能有20G片的出货量。世界各地的器件供应商纷纷亮出自己的得意之作，他们提供的器件和架构也是各具特色。业界内部可谓是百花齐放，热闹非凡，好戏不断。各行各业对单片机能力的要求也一直“得寸进尺”，而且还又要马儿跑，又要马儿不吃草——处理器必须在不怎么增加主频和功耗的条件下干更多的活儿。另一方面，处理器之间的互连也在加深，看这一串串熟悉的字眼：串口，USB，以太网，无线数传……处理器如欲支持这些数据通道，就必须在片上塞进更多的外设。软件方面的情况也如出一辙：应用程序的功能一直在花样翻新，性能需求也是变本加厉：更高的运算速度，更硬的实时能力，更多的功能模块，更炫的图形界面，……所有这些要求单片机都得照单全收。在这个大环境下，ARM Cortex-M3处理器，作为Cortex系列的处女作，为了让32位处理器入主作庄单片机市场，轰轰烈烈地诞生了！由于采用了最新的设计技术，它的门数更低，性能却更强。许多曾经只能求助于高级32位处理器或DSP的软件设计，都能在CM3上跑得很快很欢。相信用不了多久，CM3就一定能在32位嵌入式处理器市场中脱颖而出，像当年8051推动整个业界那样，再次放飞设计师的梦想，实现多年的夙愿！

CM3的招牌功夫包括：

- 性能强劲。在相同的主频下能做处理更多的任务，全力支持劲爆的程序设计。
- 功耗低。延长了电池的寿命——这简直就是便携式设备的命门（如无线网络应用）。
- 实时性好。采用了很前卫甚至革命性的设计理念，使它能极速地响应中断，而且响应中断所需的周期数是确定的。
- 代码密度得到很大改善。一方面力挺大型应用程序，另一方面为低成本设计而省吃俭用。
- 使用更方便。现在从8位/16位处理器转到32位处理器之风刮得越来越猛，更简单的编程模型和更透彻的调试系统，为与时俱进的人们大大减负。
- 低成本的整体解决方案。让32位系统比和8位/16位的还便宜，低端的Cortex-M3单片机甚至还卖不到1美元。
- 遍地开花的优秀开发工具。免费的，便宜的，全能的，要什么有什么。

基于Cortex-M3内核的处理器已渐成气候，以处处满溢的先进特性力压群芳。而且架构

师们还在不停地求索降低成本的出路，同时很多组织也在尝试着实现"器件聚合"（device aggregation），使一个单一的小强芯片可以抵得上以前3、4块传统的单片机。

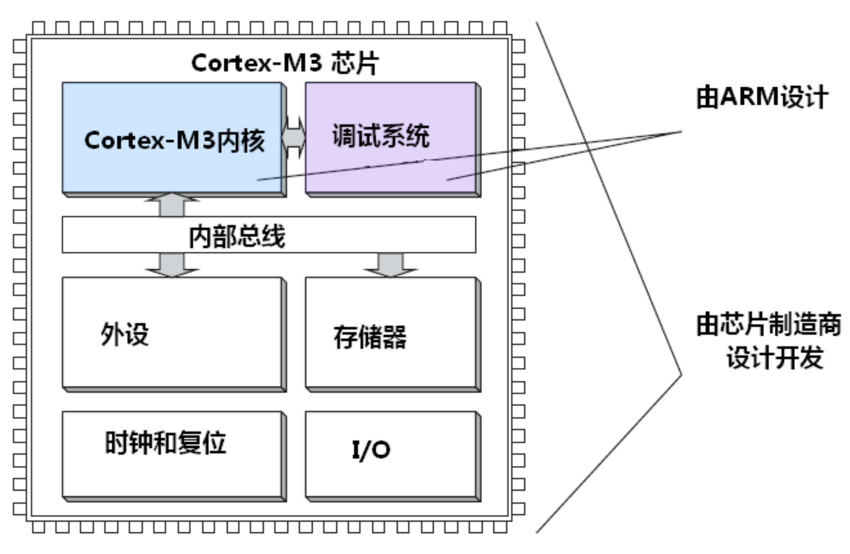
降低成本还有一招，就是使基础代码在所有系统中都可以重用，至少要方便移植。CM3的内核架构非常精工细作，使它与C语言成为了一个梦幻绝配。优质的C程序代码三下五除二就可以移植并重用，使升级和移植一下子从拦路虎变成了纸老虎。

值得一提的是，CM3并不是第一个被拿去做万金油型处理器的内核。那廉颇虽老却依然骁勇的ARM7/ARM9处理器，在通用嵌入式处理器市场中德高望重，至今拥有无数铁杆粉丝。半导体业界的群英们，像NXP（philips）、TI、Atmel、OKI、ST等，都以ARM为内核，做出了各自身怀绝技的32位MCU。ARM7作为最受欢迎的32位嵌入式处理器，被载入了亮煌煌的几页史册——每年超过10亿片出货量，为各行各业的嵌入式设备中都找得到它们的身影。

CM3作为ARM7的后继者，大刀阔斧地改革了设计架构。从而显著地简化了编程和调试的复杂度，处理能力也更加强大。除此之外，CM3还突破性地引入了很多时尚的甚至崭新的技术，专门满足单片机应用程序的需求。比如，服务于"使命-关键"应用的不可屏蔽中断，极度敏捷并且拥有确定性的嵌套向量中断系统，原子性质的位操作，还有一个可选的内存保护单元。这些令人惊艳和振奋的新特性，让老的ARM玩家们再次找到"初恋"时烈焰迸发的感觉，也使萍水相逢就有激爽触电般的体验！相信读者一旦有机会用到了它，就会为它的秀外慧中而赞叹，爱不释手！

Cortex-M3 处理器内核 vs. 基于Cortex-M3的MCU

Cortex-M3处理器内核是单片机的中央处理单元（CPU）。完整的基于CM3的MCU还需要很多其它组件。在芯片制造商得到CM3处理器内核的使用授权后，它们就可以把CM3内核用在自己的硅片设计中，添加存储器，外设，I/O以及其它功能块。不同厂家设计出的单片机会有不同的配置，包括存储器容量、类型、外设等都各具特色。本书主讲处理器内核本身。如果想要了解某个具体型号的处理器，还需查阅相关厂家提供的文档。



ARM及ARM架构的背景

一路走来

让我们回顾一下ARM的进化史，你会知道为什么会有品种如此之多的ARM处理器和ARM架构。

ARM在1990年成立，当初的名字是"Advanced RISC Machines Ltd.,"，当时它是三家公司的

合资——它们分别是苹果电脑，Acorn电脑公司，以及VLSI技术（公司）。在1991年，ARM推出了ARM6处理器家族，VLSI则是第一个吃螃蟹的人。后来，陆续有其它巨头：包括TI, NEC, Sharp, ST等，都获取了ARM授权，它们真正地把ARM处理器大面积地铺开，使得ARM处理器在手机，硬盘控制器，PDA，家庭娱乐系统以及其它消费电子中都大展雄才。

现如今，ARM芯片的出货量每年都比上一年多20亿片以上。不像很多其它的半导体公司，ARM从不制造和销售具体的处理器芯片。取而代之的，是ARM把处理器的设计授权给相关的商务合作伙伴，让他们去根据自己的强项设计具体的芯片，这些伙伴可都是巨头啊。基于ARM低成本和高效的处理器设计方案，得到授权的厂商生产了多种多样的的处理器、单片机以及片上系统(SoC)。这种商业模式就是所谓的"知识产权授权"。

除了设计处理器，ARM也设计系统级IP和软件IP。为了挺它们，ARM开发了许多配套的基础开发工具、硬件以及软件产品。使用这些工具，合作伙伴可以更加舒心地开发他们自己的产品。

ARM的各种架构版本

ARM十几年如一日地开发新的处理器内核和系统功能块。这些包括流行的ARM7TDMI处理器，还有更新的高档产品ARM1176TZ(F)-S处理器，后者能拿去做高档手机。功能的不断进化，处理水平的持续提高，年深日久造就了一系列的ARM架构。要说明的是，架构版本号和名字中的数字并不是一码事。比如，ARM7TDMI是基于ARMv4T架构的（T表示支持"Thumb指令"）；ARMv5TE架构则是伴随着ARM9E处理器家族亮相的。ARM9E家族成员包括ARM926E-S和ARM946E-S。ARMv5TE架构添加了"服务于多媒体应用增强的DSP指令"。

后来又出了ARM11，ARM11是基于ARMv6架构建成的。基于ARMv6架构的处理器包括ARM1136J(F)-S，ARM1156T2(F)-S，以及ARM1176JZ(F)-S。ARMv6是ARM进化史上的一个重要里程碑：从那时候起，许多突破性的新技术被引进，存储器系统加入了很多的崭新的特性，单指令流多数据流（SIMD）指令也是从v6开始首次引入的。而最前卫的新技术，就是经过优化的Thumb-2指令集，它专为低成本的单片机及汽车组件市场。

ARMv6的设计中还有另一个重大的决定：虽然这个架构要能上能下，从最低端的MCU到最高端的"应用处理器"都通吃，但不能因此就这也会，那也会，但就是都不精。仍须定位准确，使处理器的架构能胜任每个应用领域。结果就是，要使ARMv6能够灵活地配置和剪裁。对于成本敏感市场，要设计一个低门数的架构，让她有极强的确定性；另一方面，在高端市场上，不管是要有丰富功能的还是要有高性能的，都要有拿得出手的好东西。

最近的几年，基于从ARMv6开始的新设计理念，ARM进一步扩展了它的CPU设计，成果就是ARMv7架构的闪亮登场。在这个版本中，内核架构首次从单一款式变成3种款式。

- 款式A：设计用于高性能的"开放应用平台"——越来越接近电脑了
- 款式R：用于高端的嵌入式系统，尤其是那些带有实时要求的——又要快又要实时。
- 款式M：用于深度嵌入的，单片机风格的系统中——本书的主角。

让我们再进距离地考察这3种款式：

- 款式A（ARMv7-A）：需要运行复杂应用程序的"应用处理器"[译注1]。支持大型嵌入式操作系统（不一定实时——译注），比如Symbian（诺基亚智能手机用），Linux，以及微软的Windows CE和智能手机操作系统Windows Mobile。这些应用需要劲爆的处理性能，并且需要硬件MMU实现的完整而强大的虚拟内存机制，还基本上会配有Java支持，有时还要求一个安全程序执行环境（用于电子商务——译注）。典型的产品包括高端手机和手持仪器，电子钱包以及金融事务处理机。

［译注1］：这里的"应用"尤指大型应用程序，像办公软件，导航软件，网页浏览器等。这些软件的使用习惯和开发模式都很像PC上的软件，但是基本上没有实时要求。

- 款式R（ARMv7-R）：硬实时且高性能的处理器。标的是高端实时[注1]市场。那些高级的玩意，像高档轿车的组件，大型发电机控制器，机器手臂控制器等，它们使用的处理器不但要很好很强大，还要极其可靠，对事件的反应也要极其敏捷。
- 款式M（ARMv7-M）：认准了旧世代单片机的应用而量身定制。在这些应用中，尤其是对于实时控制系统，低成本、低功耗、极速中断反应以及高处理效率，都是至关重要的。Cortex系列是v7架构的第一次亮相，其中Cortex-M3就是按款式M设计的。

［注1］：通用处理器能否胜任实时系统的控制，常遭受质疑，并且在这方面的争论从没停止过。从定义的角度讲，"实时"就是指系统必须在给定的死线（deadline，亦称作"最后期限"）内做出响应。在一个以ARM处理器为核心的系统中，决定能否达到"实时"这个目标的，有很多因素，包括是否使用"实时操作系统"，中断延迟，存储器延时，以及当时处理器是否在运行更高优先级的中断服务例程。

本书认准了Cortex-M3就一猛子扎下去。到目前为止，Cortex-M3也是款式M中被抚养成人的独苗。其它Cortex家族的处理器包括款式A的Cortex-A8（应用处理器），款式R的Cortex-R4（实时处理器）。

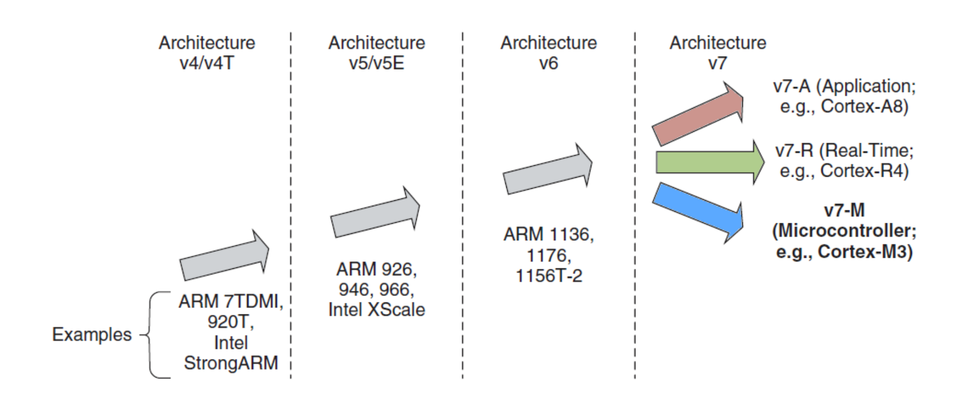


图1.2 ARM处理器架构进化史

ARMv7-M的私房秘密都记录在《The ARMv7-M Architecture Application Level Reference Manual》中（本书也讲了很多"System Level"的内容——译注），ARM已经将其公开。《Cortex M3 Technical Reference Manual》中则记录了实现v7-M时的很多细节和花絮。又因为v7M中列出的指令有一些是可选的，而CM3裁掉了一部分，所以在这个文档中重新列出了被CM3支持的指令集。

处理器命名法

以前，ARM使用一种基于数字的命名法。在早期（1990s），还在数字后面添加字母后缀，用来进一步明细该处理器支持的特性。就拿ARM7TDMI来说，T代表Thumb指令集，D是说支持JTAG调试(Debugging)，M意指快速乘法器，I则对应一个嵌入式ICE模块。后来，这4项基本功能成了任何新产品的标配，于是就不再使用这4个后缀——相当于默许了。但是新的后缀不断加入，包括定义存储器接口的，定义高速缓存的，以及定义"紧耦合存储器（TCM）"的，于是形成了新一套命名法，这套命名法也是一直在使用的。

表1.1 ARM处理器名字

处理器名字	架构版本号	存储器管理特性	其它特性
ARM7TDMI	v4T		
ARM7TDMI-S	v4T		
ARM7EJ-S	v5E		DSP,Jazelle[译注3]
ARM920T	v4T	MMU	
ARM922T	v4T	MMU	
ARM926EJ-S	v5E	MMU	DSP,Jazelle
ARM946E-S	v5E	MPU	DSP
ARM966E-S	v5E		DSP
ARM968E-S	v5E		DMA,DSP
ARM966HS	v5E	MPU（可选）	DSP
ARM1020E	v5E	MMU	DSP
ARM1022E	v5E	MMU	DSP
ARM1026EJ-S	v5E	MMU 或 MPU[译注2]	DSP, Jazelle
ARM1136J(F)-S	v6	MMU	DSP, Jazelle
ARM1176JZ(F)-S	v6	MMU+TrustZone	DSP, Jazelle
ARM11 MPCore	v6	MMU+多处理器缓存支持	DSP
ARM1156T2(F)-S	v6	MPU	DSP
Cortex-M3	v7-M	MPU（可选）	NVIC
Cortex-R4	v7-R	MPU	DSP
Cortex-R4F	v7-R	MPU	DSP+浮点运算
Cortex-A8	v7-A	MMU+TrustZone	DSP, Jazelle

［译注2］：Jazelle是ARM处理器的硬件Java加速器。

［译注3］：MMU，存储器管理单元，用于实现虚拟内存和内存的分区保护，这是应用处理器与嵌入式处理器的分水岭。电脑和数码产品所使用的处理器几乎清一色地都带MMU。但是MMU也引入了不确定性，这有时是嵌入式领域——尤其是实时系统不可接受的。然而对于安全关键（safety-critical）的嵌入式系统，还是不能没有内存的分区保护的。为解决矛盾，于是就有了MPU。可以把MPU认为是MMU的功能子集，它只支持分区保护，不支持具有"定位决定性"的虚拟内存机制。

到了架构7时代，ARM改革了一度使用的，冗长的、需要"解码"的数字命名法，转到另一种看起来比较整齐的命名法。比如，ARMv7的三个款式都以Cortex作为主名。这不仅更加澄清并且"精装"了所使用的ARM架构，也避免了新手对架构号和系列号的混淆。例如，ARM7TDMI并不是一款ARMv7的产品，而是辉煌起点——v4T架构的产品。

指令系统的开发

为了增强和扩展指令系统的能力而奋斗，多少年来这一直是ARM锲而不舍的精神动力。

由于历史原因（从ARM7TDMI开始），ARM处理器一直支持两种形式上相对独立的指令集，它们分别是：

- 32位的ARM指令集。对应处理器状态：ARM状态

- 16位的Thumb指令集。对应处理器状态：Thumb状态

可见，这两种指令集也对应了两种处理器执行状态。在程序的执行过程中，处理器可以动态地在两种执行状态之中切换。实际上，Thumb指令集在功能上是ARM指令集的一个子集，但它能带来更高的代码密度，给目标代码减肥。这对于要勒紧裤腰带的应用还是很经济的。

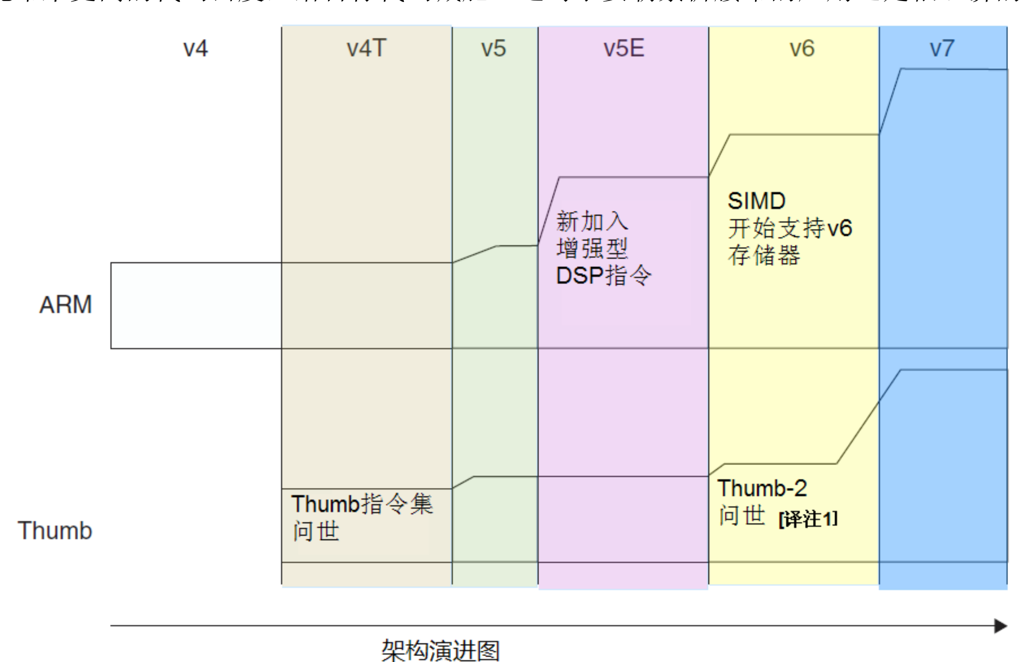


图1.3 指令集演进图

译注1: 原书把Thumb-2的问世时间放到v7中，但根据其它权威文献的记录，似有误，应在v6中问世。（如《ARM and Thumb-2 Instruction Set Quick Reference Card》中的描述）

随着架构版本号的更新，新好指令不断地加入ARM和Thumb指令集中。附录2中给出的内容，就是Thumb指令在架构进化过程中的改变记录。Thumb-2是2003年盛夏的果实，它是Thumb的超集，它支持both 16位和32位指令。

指令集的详细说明在《The ARM Architecture Reference Manual》（简称为ARMARM）中。每次ARM出新版本时此手册都有更新。到了v7时，因为以前的单一架构被分成了3个款式，这个规格书也就跟着变成了3本。为Corex-M3的ARMv7-M架构而写的那本叫《ARMv7-M Architecture Application Level Reference Manual(Ref2)》，对于软件开发人员，那里面把该说的都说了。

Thumb-2指令集体系体系结构（ISA）

Thumb-2真不愧是一个突破性的指令集。它强大，它易用，它轻佻，它高效。Thumb-2是16位Thumb指令集的一个超集，在Thumb-2中，16位指令首次与32位指令并存，结果在Thumb状态下可以做的事情一下子丰富了许多，同样工作需要的指令周期数也明显下降。

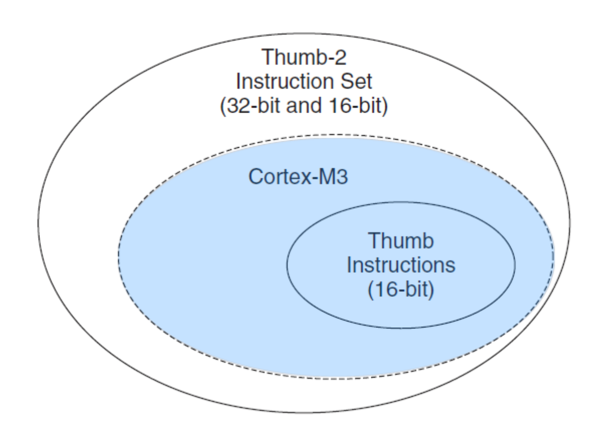


图1.4 Thumb-2指令集与Thumb指令集的关系

从图中可见，Cortex-M3勇敢地拒绝了32位ARM指令集，却把自己的处理能力以身相许般地全托给Thumb-2指令集。这可能有些令人意外，但事实上这却见证了Cortex-M3的用情专一：在内核水平上，就已经为适应单片机和小内存器件而抉择、取舍过了。但她没有嫁错郎，因为Thumb-2完全胜任在这个领域挑大梁。不过，这也意味着Cortex-M3作为新生代处理器，不是向后兼容的。因此，为ARM7写的ARM汇编语言程序不能直接移植到CM3上来。不过，CM3支持绝大多数传统的Thumb指令，因此用Thumb指令写的汇编程序就从善如流了。

在支持了both 16位和32位指令之后，就无需烦心地把处理器状态在Thumb和ARM之间来回的切换了。这种事在ARM7和ARM9是司空见惯的，尤其是在使用大型条件嵌套，以及执行复杂运算的时候，能精妙地游走于不同状态之间，那可是当年要成为大虾的基本功。

Cortex-M3是ARMv7架构的掌上明珠。和曾经红透整个业界的老一辈ARM7相比，Cortex-M3则是新生代的偶像，处处闪耀着青春的光芒活力。比如，硬件除法器被带到CM3中；乘法方面，也有好几条新指令闪亮登场，用于提升data-crunching的性能。CM3的出现，还在ARM处理器中破天荒地支持了“非对齐数据访问支持”。

Cortex-M3处理器的舞台

高性能＋高代码密度＋小硅片面积，3璧合一，使得CM3大面积地成为理想的处理平台：

- 低成本单片机：CM3与生俱来就适合做单片机，甚至简单到用于做玩具和小电器的单片机，都能使用CM3作为内核。这里本是8位机和16位机统治最牢固的腹地，但是CM3更便宜，更高性能，更易使用，所以值得开发者们转到这个新生的ARM32位系统中来，哪怕花点时间重新学习。
- 汽车电子：CM3也是汽车电子的好侏。CM3同时拥有非常高的性能和极低的中断延迟，打入实时领域的大门。CM3处理器能支持多达240个外部中断，内建了嵌套向量中断控制器，还可以选择配上一个存储器保护单元（MPU）。所有这些，使它用于高集成度低成本的汽车应用最合适不过了。
- 数据通信：CM3的低成本＋高效率，再加上Thumb-2的强大位操作指令s，使CM3非常理想地适合于很多数据通信应用，尤其是无线数传和Ad-Hoc网络，如ZigBee和蓝牙等。
- 工业控制：在工控场合，关键的要素在于简洁、快速响应以及可靠。再一次地，CM3处理器的中断处理能力，低中断延迟，强化的故障处理能力（fault-handing，以后fault就不再译成中文了——译注），足以让它能昂首挺胸地踏入这片热土。
- 消费类产品：以往，在许多消费产品中，都必须使用一块甚至好几块高性能的微处理器。

你别看CM3只是个小处理器，它的高性能和MPU机制可是足以让复杂的软件跑起来的，同时提供健壮的存储器保护。

目前在市场上已经有了好多基于Cortex-M3内核的处理器产品，最便宜的还不到1美元，让ARM终于比很多8位机还便宜了。

本书的组织

Chpt 1和2，	Cortex-M3的介绍和概览
Chpt 3-6，	Cortex-M3的基础知识
Chpt 7-9，	异常与中断
Chpt 10和11，	论述在Cortex-M3的编程
Chpt 12-14，	Cortex-M3的硬件特性
Chpt 15-16，	Cortex-M3的调试支持
Chpt 17-20，	在Cortex-M3上的应用软件开发
附录s	

深入研究用的读物

本书并没有面面俱到地谈及Cortex-M3的技术细节。本书靠前的章节用来做Cortex-M3新手的敲门砖，同时也是CM3处理器的增值参考资料。如果要进一步地学习，就需要从ARM网站下载下面这些重量级的权威资料：

《The Cortex-M3 Technical Reference Manual》，深入了处理器的内心，编程模型，存储器映射，还包括了指令时序。

《 The ARMv7-M Architecture Application Level Reference Manual》第2版，对指令集和存储器模型都提供了最不嫌繁的说明。

其它半导体厂家提供的，基于CM3单片机的数据手册。

如想了解更多总线协议的细节，可以去看《AMBA Specification 2.0》（第4版），它讲了更多AMBA接口的内幕。

对于C程序员，可以从《ARM Application Note 179: Cortex-M3 Embedded Software Development》（第7版）中得到一些编程技巧和提示。

本书假设你已经涉足过嵌入式编程，有一些基本知识和经验。如果你是位产品经理或者是想先浅浅地尝一尝，请先读第2章，试着找找感觉再决定要不要深入学习。这一章浓缩了全书的精华，走马观花地讲了Cortex-M3内核。

第2章

Cortex-M3概览

内容提要：

- 简介
- 寄存器组
- 操作模式和特权极别
- 内建的嵌套向量中断控制器
- 存储器映射
- 总线接口
- 存储器保护单元
- 指令系统
- 中断和异常
- 调试支持
- 小结

简介

Cortex-M3 是一个 32 位处理器内核。内部的数据路径是 32 位的，寄存器是 32 位的，存储器接口也是 32 位的。CM3 采用了哈佛结构，拥有独立的指令总线和数据总线，可以让取指与数据访问并行不悖。这样一来数据访问不再占用指令总线，从而提升了性能。为实现这个特性， CM3 内部含有好几条总线接口，每条都为自己的应用场合优化过，并且它们可以并行工作。但是另一方面，指令总线和数据总线共享同一个存储器空间（一个统一的存储器系统）。换句话说，不是因为有两条总线，可寻址空间就变成 8GB 了。

比较复杂的应用可能需要更多的存储系统功能，为此 CM3 提供一个可选的 MPU，而且在需要的情况下也可以使用外部的 cache。另外在 CM3 中，Both 小端模式和大端模式都是支持的。

CM3 内部还附赠了好多调试组件，用于在硬件水平上支持调试操作，如指令断点，数据观察点等。另外，为支持更高级的调试，还有其它可选组件，包括指令跟踪和多种类型的调试接口。

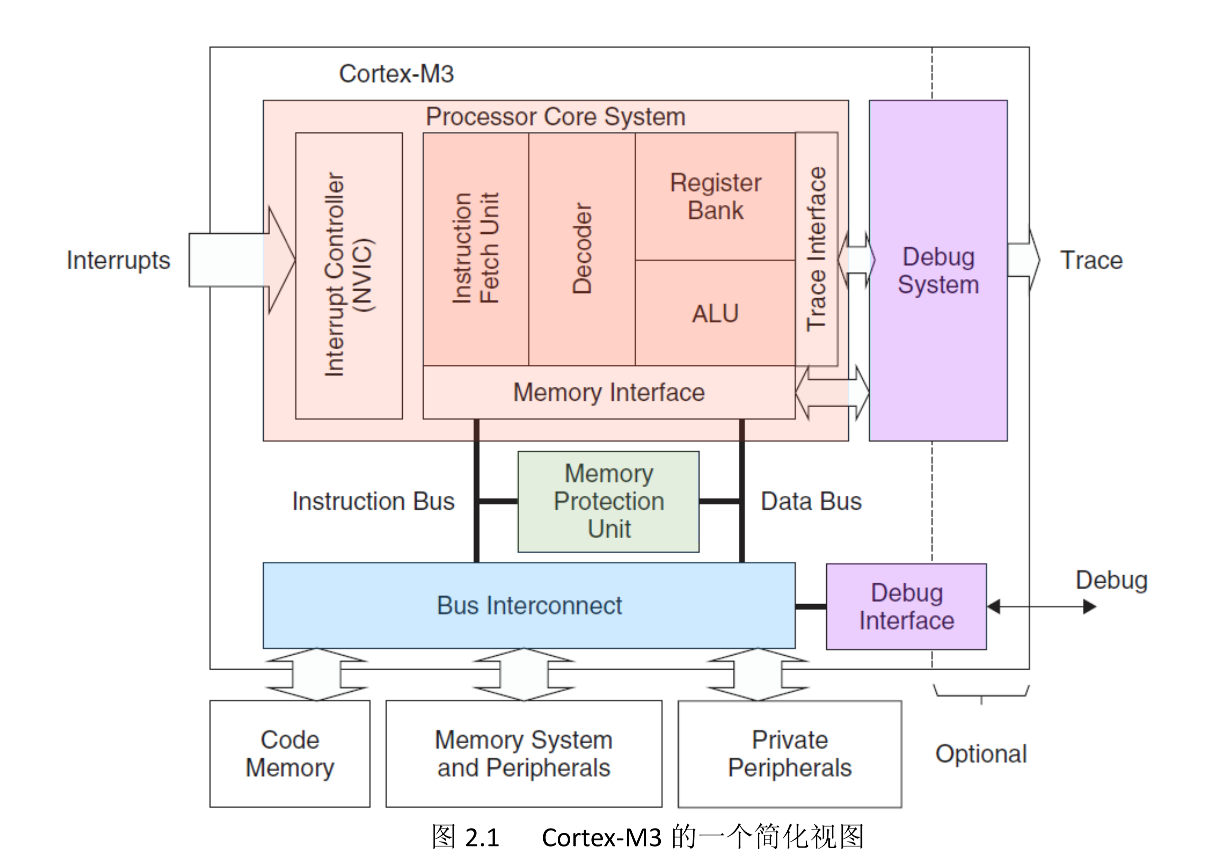
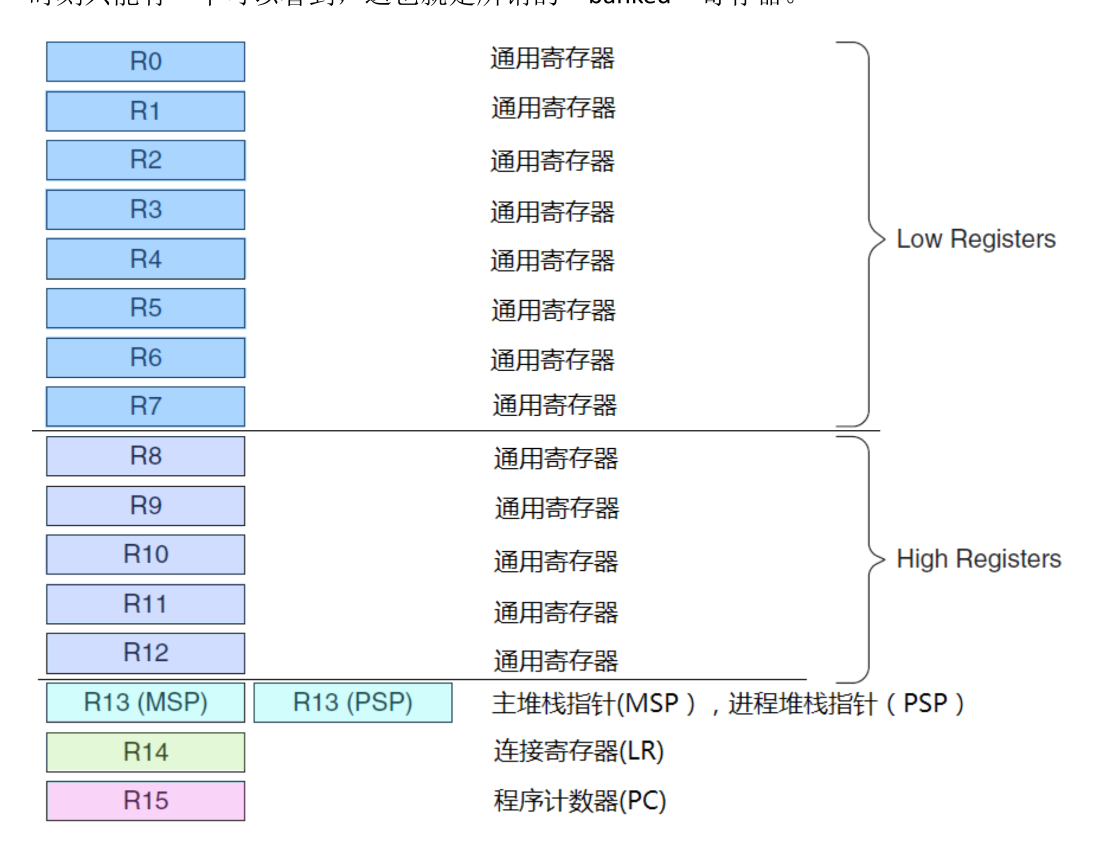


图 2.1 Cortex-M3 的一个简化视图

寄存器组

Cortex-M3 处理器拥有 R0-R15 的寄存器组。其中 R13 作为堆栈指针 SP。SP 有两个，但在同一时刻只能有一个可以看到，这也就是所谓的“banked”寄存器。



R0-R12：通用寄存器

R0-R12 都是 32 位通用寄存器，用于数据操作。但是注意：绝大多数 16 位 Thumb 指令只能访问 R0-R7，而 32 位 Thumb-2 指令可以访问所有寄存器。

Banked R13: 两个堆栈指针

Cortex-M3 拥有两个堆栈指针，然而它们是 banked，因此任一时刻只能使用其中的一个。

- 主堆栈指针（MSP）：复位后缺省使用的堆栈指针，用于操作系统内核以及异常处理例程（包括中断服务例程）
- 进程堆栈指针（PSP）：由用户的应用程序代码使用。

堆栈指针的最低两位永远是 0，这意味着堆栈总是 4 字节对齐的。

在 ARM 编程领域中，凡是打断程序顺序执行的事件，都被称为异常(exception)。除了外部中断外，当有指令执行了“非法操作”，或者访问被禁的内存区间，因各种错误产生的 fault，以及不可屏蔽中断发生时，都会打断程序的执行，这些情况统称为异常。在不严格的上下文中，异常与中断也可以混用。另外，程序代码也可以主动请求进入异常状态的（常用于系统调用）。

R14：连接寄存器

当呼叫一个子程序时，由 R14 存储返回地址

不像大多数其它处理器，ARM 为了减少访问内存的次数（访问内存的操作往往要 3 个以上指令周期，带 MMU 和 cache 的就更加不确定了），把返回地址直接存储在寄存器中。这样足以使很多只有 1 级子程序调用的代码无需访问内存（堆栈内存），从而提高了子程序调用的效率。如果多于 1 级，则需要把前一级的 R14 值压到堆栈里。在 ARM 上编程时，应尽量只使用寄存器保存中间结果，迫不得以时才访问内存。在 RISC 处理器中，为了强调访内操作越过了处理器的界线，并且带来了对性能的不利影响，给它取了一个专业的术语：溅出。

R15：程序计数寄存器

指向当前的程序地址。如果修改它的值，就能改变程序的执行流（很多高级技巧就在这里面——译注）。

特殊功能寄存器

Cortex-M3 还在内核水平上搭载了若干特殊功能寄存器，包括

程序状态字寄存器组（PSRs）

中断屏蔽寄存器组（PRIMASK, FAULTMASK, BASEPRI）

控制寄存器（CONTROL）

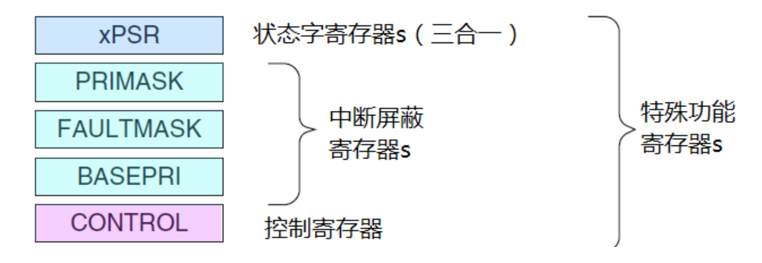


图 2.3： Cortex-M3 中的特殊功能寄存器集合

表 2.1　寄存器及其功能

寄存器	功能
xPSR	记录 ALU 标志（0 标志，进位标志，负数标志，溢出标志），执行状态，以及当前正服务的中断号
PRIMASK	除能所有的中断——当然了，不可屏蔽中断（NMI）才不甩它呢。
FAULTMASK	除能所有的 fault——NMI 依然不受影响，而且被除能的 faults 会“上访”，见后续章节的叙述。
BASEPRI	除能所有优先级不高于某个具体数值的中断。
CONTROL	定义特权状态（见后续章节对特权的叙述），并且决定使用哪一个堆栈指针

第 3 章对此有展开的叙述。

操作模式和特权级别

Cortex-M3 处理器支持两种处理器的操作模式，还支持两级特权操作。

两种操作模式分别为：处理者模式(handler mode，以后不再把 handler 中译——译注)和线程模式（thread mode）。引入两个模式的本意，是用于区别普通应用程序的代码和异常服务例程的代码——包括中断服务例程的代码。

Cortex-M3 的另一个侧面则是特权的分级——特权级和用户级。这可以提供一种存储器访问的保护机制，使得普通的用户程序代码不能意外地，甚至是恶意地执行涉及到要害的操作。处理器支持两种特权级，这也是一个基本的安全模型。

	特权级	用户级
异常handler的代码	handler模式	错误的用法
主应用程序的代码	线程模式	线程模式

图 2.4　Cortex-M3 下的操作模式和特权级别

在 CM3 运行主应用程序时（线程模式），既可以使用特权级，也可以使用用户级；但是异常服务例程必须在特权级下执行。复位后，处理器默认进入线程模式，特权极访问。在特权级下，程序可以访问所有范围的存储器(如果有 MPU，还要在 MPU 规定的禁地之外)，并且可以执行所有指令。

在特权级下的程序可以为所欲为，但也可能会把自己给玩进去——切换到用户级。一旦进入用户级，再想回来就得走“法律程序”了——用户级的程序不能简简单单地试图改写 CONTROL 寄存器就回到特权级，它必须先“申诉”：执行一条系统调用指令(SVC)。这会触发 SVC 异常，然后由异常服务例程（通常是操作系统的一部分）接管，如果批准了进入，则异常服务例程修改 CONTROL 寄存器，才能在用户级的线程模式下重新进入特权级。

事实上，从用户级到特权级的唯一途径就是异常：如果在程序执行过程中触发了一个异常，处理器总是先切换入特权级，并且在异常服务例程执行完毕退出时，返回先前的状态（也可以手工指定返回的状态——译注）。

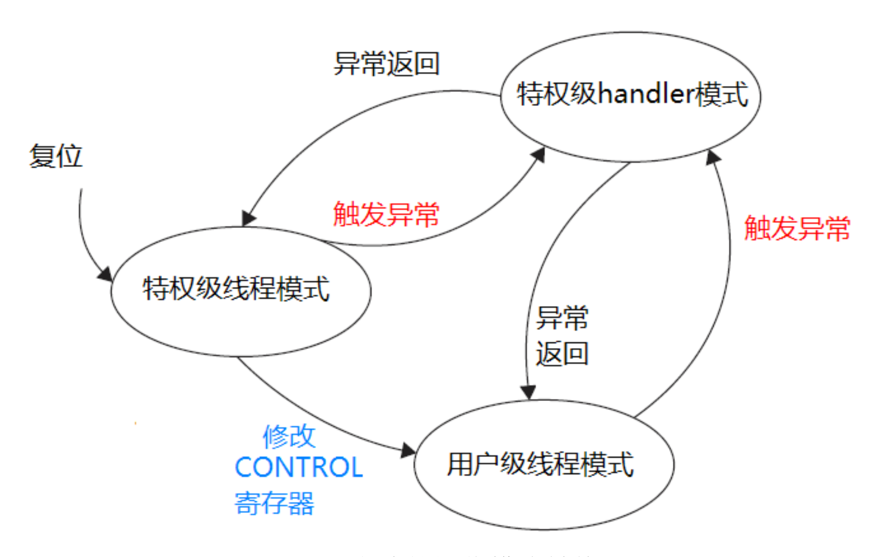


图 2.5　　合法的操作模式转换图

通过引入特权级和用户级，就能够在硬件水平上限制某些不受信任的或者还没有调试好的程序，不让它们随便地配置涉及要害的寄存器，因而系统的可靠性得到了提高。进一步地，如果配了 MPU，它还可以作为特权机制的补充——保护关键的存储区域不被破坏，这些区域通常是操作系统的区域。

举例来说，操作系统的内核通常都在特权级下执行，所有没有被 MPU 禁掉的存储器都可以访问。在操作系统开启了一个用户程序后，通常都会让它在用户级下执行，从而使系统不会因某个程序的崩溃或恶意破坏而受损。

内建的嵌套向量中断控制器

Cortex-M3 在内核水平上搭载了一颗中断控制器——嵌套向量中断控制器 NVIC(Nested Vectored Interrupt Controller)。它与内核有很深的"私交"——与内核是紧耦合的。NVIC 提供如下的功能：

- 可嵌套中断支持
- 向量中断支持
- 动态优先级调整支持
- 中断延迟大大缩短
- 中断可屏蔽

可嵌套中断支持

可嵌套中断支持的作用范围很广，覆盖了所有的外部中断和绝大多数系统异常。外在表现是，这些异常都可以被赋予不同的优先级。当前优先级被存储在 xPSR 的专用字段中。当一个异常发生时，硬件会自动比较该异常的优先级是否比当前的异常优先级更高。如果发现来了更高优先级的异常，处理器就会中断当前的中断服务例程（或者是普通程序），而服务新来的异常——即立即抢占。

向量中断支持

当开始响应一个中断后，CM3 会自动定位一张向量表，并且根据中断号从表中找出 ISR 的入口地址，然后跳转过去执行。不需要像以前的 ARM 那样，由软件来分辨到底是哪个中断发生了，也无需半导体厂商提供私有的中断控制器来完成这种工作。这么一来，中断延迟时间大为缩短。

动态优先级调整支持

软件可以在运行时期更改中断的优先级。如果在某 ISR 中修改了自己所对应中断的优先级，而且这个中断又有新的实例处于悬起中（pending），也不会自己打断自己，从而没有重入(reentry) [译注7] 风险。

［译注 7］：所谓的重入，就是指某段子程序还没有执行完，就因为中断或者是多任务操作系统的调度原因，导致该子程序在一个新的寄存器上下文中被执行（请不要把重入与递归混淆，它们有本质的区别）。这种情况常常会闹出乱子，因此有“可重入性”的研究。

中断延迟大大缩短

Cortex-M3 为了缩短中断延迟，引入了好几个新特性。包括自动的现场保护和恢复，以及其它的措施，用于缩短中断嵌套时的 ISR 间延迟。详情请见后面关于“咬尾中断”和“晚到中断”的讲述。

中断可屏蔽

既可以屏蔽优先级低于某个阈值的中断/异常 [译注 8] (设置 BASEPRI 寄存器)，也可以全体封杀(设置 PRIMASK 和 FAULTMASK 寄存器)。这是为了让时间关键（time-critical）的任务能在死线(deadline，或曰最后期限)到来前完成，而不被干扰。

［译注 8］：鉴于（外部）中断的常见性，以后译文中如果没有特殊说明，凡是提到“异常”，均指除了外部中断之外的异常，而使用“中断”来表示所有外部中断——也就是对于处理器来说是异步的中断。

存储器映射

总体来说，Cortex-M3 支持 4GB 存储空间，如图 2.6 所示地被划分成若干区域。

地址	大小	区域	说明
0xFFFFFFFF 0xE0000000	512 MB	System Level	服务于CM3的私房外设，包括NVIC寄存器，MPU寄存器以及片上调试组件
0xDFFFFFFF 0xA0000000	1GB	External Device	主要用于扩展片外的外设（像8051配8255似的）
0x9FFFFFFF 0x60000000	1GB	External RAM	用于扩展外部存储器
0x5FFFFFFF 0x40000000	512MB	Peripherals	用于片上外设
0x3FFFFFFF 0x20000000	512MB	SRAM	用于片上静态RAM
0x1FFFFFFF 0x00000000	512MB	Code	代码区。也可用于存储启动后缺省的中断向量表

从图中可见，不像其它的 ARM 架构，它们的存储器映射由半导体厂家说了算，Cortex-M3 预先定义好了“粗线条的”存储器映射。通过把片上外设的寄存器映射到外设区，就可以简单地以访问内存的方式来访问这些外设的寄存器，从而控制外设的工作。结果，片上外设可以使用 C 语言来操作。这种预定义的映射关系，也使得对访问速度可以做高度的优化，而且对于片上系统的设计而言更易集成（还有一个重要的，不用每学一种不同的单片机就要熟悉一种新的存储器映射——译注）。

Cortex-M3 的内部拥有一个总线基础设施，专用于优化对这种存储器结构的使用。在此之上，CM3 甚至还允许这些区域之间“越权使用”。比如说，数据存储器也可以被放到代码区，而且代码也能够在外部 RAM 区中执行（但是会变慢不少——译注）。

处于最高地址的系统级存储区，是 CM3 用于藏“私房钱”的——包括中断控制器、MPU 以及各种调试组件。所有这些设备均使用固定的地址（本书第 5 章讨论存储器系统）。通过把基础设施的地址定死，就至少在内核水平上，为应用程序的移植扫清了障碍。

总线接口

Cortex-M3 内部有若干个总线接口，以使 CM3 能同时取址和访内（访问内存），它们是：

- 指令存储区总线（两条）
- 系统总线
- 私有外设总线

有两条代码存储区总线负责对代码存储区的访问，分别是 I-Code 总线和 D-Code 总线。前者用于取指，后者用于查表等操作，它们按最佳执行速度进行优化。

系统总线用于访问内存和外设，覆盖的区域包括 SRAM，片上外设，片外 RAM，片外扩展设备，以及系统级存储区的部分空间。

私有外设总线负责一部分私有外设的访问，主要就是访问调试组件。它们也在系统级存储区。

存储器保护单元（MPU）

Cortex-M3 有一个可选的存储器保护单元。配上它之后，就可以对特权级访问和用户级访问分别施加不同的访问限制。当检测到犯规（violated）时，MPU 就会产生一个 fault 异常，可以由 fault 异常的服务例程来分析该错误，并且在可能时改正它。

MPU 有很多玩法。最常见的就是由操作系统使用 MPU，以使特权级代码的数据，包括操作系统本身的数据不被其它用户程序弄坏。MPU 在保护内存时是按区管理的(“区”的原文是 region，以后不再中译此名词——译注)。它可以把某些内存 region 设置成只读，从而避免了那里的内容意外被更改；还可以在多任务系统中把不同任务之间的数据区隔离。一句话，它会使嵌入式系统变得更加健壮，更加可靠（很多行业标准，尤其是航空的，就规定了必须使用 MPU 来行使保护职能——译注）。

指令集

Cortex-M3 只使用 Thumb-2 指令集。这是个了不起的突破，因为它允许 32 位指令和 16 位指令水乳交融，代码密度与处理性能两手抓，两手都硬。而且虽然它很强大，却依然易于使用。

在过去，做 ARM 开发必须处理好两个状态。这两个状态是井水不犯河水的，它们是：32 位的 ARM 状态和 16 位的 Thumb 状态。当处理器在 ARM 状态下时，所有的指令均是 32 位的（哪怕只是个”NOP”指令），此时性能相当高。而在 Thumb 状态下，所有的指令均是 16 位的，代码密度提高了一倍。不过，thumb 状态下的指令功能只是 ARM 下的一个子集，结果可能需要更多条的指令去完成相同的工作，导致处理性能下降。

为了取长补短，很多应用程序都混合使用 ARM 和 Thumb 代码段。然而，这种混合使用是有额外开销（overhead）的，时间上的和空间上的都有，主要发生在状态切换之时。另一方面，ARM 代码和 Thumb 代码需要以不同的方式编译，这也增加了软件开发管理的复杂度。

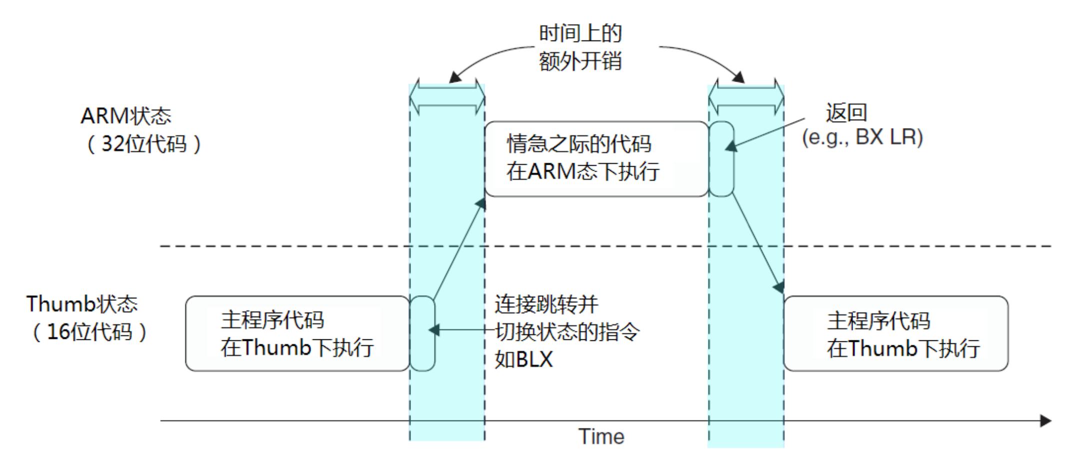


图 2.7 在诸如 ARM7 处理器上的状态切换模式图

伴随着 Thumb-2 指令集的横空出世，终于可以在单一的操作模式下搞定所有处理了，再也没有来回切换的事来烦你了。事实上，Cortex-M3 内核干脆都不支持 ARM 指令，中断也在 Thumb 态下处理（以前的 ARM 总是在 ARM 状态下处理所有的中断和异常）。这可不是小便宜，它使 CM3 在好几个方面都比传统的 ARM 处理器更先进：

- 消灭了状态切换的额外开销，节省了 both 执行时间和指令空间。
- 不再需要把源代码文件分成按 ARM 编译的和按 Thumb 编译的，软件开发的管理大大减负。
- 无需再反复地求证和测试：究竟该在何时何地切换到何种状态下，我的程序才最有效率。开发

软件容易多了。

不少有趣和强大的指令为 Cortex-M3 注入了新鲜的青春血液，下面给出几个例子：

- UBFX，BFI，BFC：位段提取，位段插入，位段清零。支持 C 位段，也简化了外设寄存器操作。
- CLZ，RBIT： 计算前导零指令和位反转指令。二者组合使用能实现一些特技
- UDIV，SDIV： 无符号除法和带符号除法指令。
- SEV，WFE，WFI：发送事件，等待事件以及等待中断指令。用于实现多处理器之间的任务同步，还可以进入不同的休眠模式。
- MSR,MRS： 通向禁地——访问特殊功能寄存器。

因为 CM3 专情于最新的 Thumb-2，旧的应用程序需要移植和重建。对于大多数 C 源程序，只需简单地重新编译就能重建，汇编代码则可能需要大面积地修改和重写，才能使用 CM3 的新功能，并且融入 CM3 新引入的统一汇编器框架(unified assembler framework)中。

请注意：CM3 并不支持所有的 Thumb-2 指令，ARMv7-M 的规格书只要求实现 Thumb-2 的一个子集。举例来说，协处理器指令就被裁掉了（可以使用外部的数据处理引擎来替代）。CM3 也没有实现 SIMD 指令集。旧世代的一些 Thumb 指令不再需要，因此也被排除。不支持指令还包括 v6 中引入的 SETEND 指令。如欲查出一个完整的指令列表，可以去看附录 A。

中断和异常

ARMv7-M 开创了一个全新的异常模型，CM3 采用了它。请你一定要划清界线：这种异常模型跟传统 ARM 处理器使用的完全是两码事。新的异常模型“使能”了非常高效的异常处理。它支持 16-4-1=11 种系统异常（保留了 4+1 个档位），外加 240 个外部中断输入。在 CM3 中取消了 FIQ 的概念（v7 前的 ARM 都有这个 FIQ，快中断请求），这是因为有了更新更好的机制——中断优先级管理以及嵌套中断支持，它们被纳入 CM3 的中断管理逻辑中。因此，支持嵌套中断的系统就更容易实现 FIQ。

CM3 的所有中断机制都由 NVIC 实现。除了支持 240 条中断之外，NVIC 还支持 16-4-1=11 个内部异常源，可以实现 fault 管理机制。结果，CM3 就有了 256 个预定义的异常类型，如表 2.2 所示。

表 2.2 Cortex-M3 异常类型

编号	**类型**	**优先级**	**简介**
0	N/A	N/A	没有异常在运行
1	复位	-3（最高）	复位
2	NMI	-2	不可屏蔽中断（来自外部 NMI 输入脚）
3	硬(hard) fault	-1	所有被除能的 fault，都将“上访”成硬 fault
4	MemManage fault	可编程	存储器管理 fault，MPU 访问犯规以及访问非法位置
5	总线 fault	可编程	总线错误（预取流产（Abort）或数据流产）
6	用法(usage) Fault	可编程	由于程序错误导致的异常
7-10	保留	N/A	N/A
11	SVCall	可编程	系统服务调用
12	调试监视器	可编程	调试监视器（断点，数据观察点，或者是外部调试请求
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的“可悬挂请求”（pendable request）
15	SysTick	可编程	系统滴答定时器（也就是周期性溢出的时基定时器——译注）

16	IRQ #0	可编程	外中断#0
17	IRQ #1	可编程	外中断#1
…	…	…	…
255	IRQ #239	可编程	外中断#239

虽然 CM3 是支持 240 个外中断的，但具体使用了多少个是由芯片生产商决定。CM3 还有一个 NMI（不可屏蔽中断）输入脚。当它被置为有效（assert）时，NMI 服务例程会无条件地执行。

调试支持

Cortex-M3 在内核水平上搭载了若干种调试相关的特性。最主要的就是程序执行控制，包括停机(halting)、单步执行(stepping)、指令断点、数据观察点、寄存器和存储器访问、性能速写（profiling）以及各种跟踪机制。

Cortex-M3 的调试系统基于 ARM 最新的 CoreSight 架构。不同于以往的 ARM 处理器，内核本身不再含有 JTAG 接口。取而代之的，是 CPU 提供称为"调试访问接口(DAP)"的总线接口。通过这个总线接口，可以访问芯片的寄存器，也可以访问系统存储器，甚至是在内核运行的时候访问！对此总线接口的使用，是由一个调试端口(DP)设备完成的。DPs 不属于 CM3 内核，但它们是在芯片的内部实现的。目前可用的 DPs 包括 SWJ-DP(既支持传统的 JTAG 调试，也支持新的串行线调试协议)，另一个 SW-DP 则去掉了对 JTAG 的支持。另外，也可以使用 ARM CoreSignt 产品家族的 JTAG-DP 模块。这下就有 3 个 DPs 可以选了，芯片制造商可以从中选择一个，以提供具体的调试接口（通常都是选 SWJ-DP）。

此外，CM3 还能挂载一个所谓的"嵌入式跟踪宏单元（ETM）"。ETM 可以不断地发出跟踪信息，这些信息通过一个被称为"跟踪端口接口单元（TPIU）"的模块而送到内核的外部，再在芯片外面使用一个"跟踪信息分析仪"，就可以把 TIPU 输出的"已执行指令信息"捕捉到，并且送给调试主机——也就是 PC。

在 Cortex-M3 中，调试动作能由一系列的事件触发，包括断点，数据观察点，fault 条件，或者是外部调试请求输入的信号。当调试事件发生时，Cortex-M3 可能会停机，也可能进入调试监视器异常 handler。具体如何反应，则根据与调试相关寄存器的配置。

与调试相关的还有其它的绝活。现在要介绍的是"指令追踪宏单元（ITM）"，它也有自己的办法把数据送往调试器。通过把数据写到 ITM 的寄存器中，调试器能够通过跟踪接口来收集这些数据，并且显示或者处理它。此法不但容易使用，而且比 JTAG 的输出速度更快。

所有这些调试组件都可以由 DAP 总线接口来控制，CM3 内核提供 DAP 接口。此外，运行中的程序也能控制它们。所有的跟踪信息都能通过 TPIU 来访问到。

Cortex-M3 的品性简评

讲了这么多，究竟是拥有了什么，使 Cortex-M3 成为如此有突破性的新生代处理器？Cortex-M3 到底在哪里先进了？本节就给出一个小小的简评。

高性能

- 许多指令都是单周期的——包括乘法相关指令。并且从整体性能上，Cortex-M3 比得过绝大多数其它的架构。
- 指令总线和数据总线被分开，取值和访内可以并行不悖
- Thumb-2 的到来告别了状态切换的旧世代，再也不需要花时间来切换于 32 位 ARM 状态和 16 位 Thumb 状态之间了。这简化了软件开发和代码维护，使产品面市更快。

- Thumb-2 指令集为编程带来了更多的灵活性。许多数据操作现在能用更短的代码搞定，这意味着 Cortex-M3 的代码密度更高，也就对存储器的需求更少。
- 取指都按 32 位处理。同一周期最多可以取出两条指令，留下了更多的带宽给数据传输。
- Cortex-M3 的设计允许单片机高频运行（现代半导体制造技术能保证 100MHz 以上的速度）。即使在相同的速度下运行，CM3 的每指令周期数(CPI)也更低，于是同样的 MHz 下可以做更多的工作；另一方面，也使同一个应用在 CM3 上需要更低的主频。

先进的中断处理功能

- 内建的嵌套向量中断控制器支持多达 240 条外部中断输入。向量化的中断功能剧烈地缩短了中断延迟，因为不再需要软件去判断中断源。中断的嵌套也是在硬件水平上实现的，不需要软件代码来实现。
- Cortex-M3 在进入异常服务例程时，自动压栈了 R0-R3, R12, LR, PSR 和 PC，并且在返回时自动弹出它们，这多清爽！既加速了中断的响应，也再不需要汇编语言代码了（第 8 章有详述）。
- NVIC 支持对每一路中断设置不同的优先级，使得中断管理极富弹性。最粗线条的实现也至少要支持 8 级优先级，而且还能动态地被修改。
- 优化中断响应还有两招，它们分别是“咬尾中断机制”和“晚到中断机制”。
- 有些需要较多周期才能执行完的指令，是可以被中断－继续的——就好比它们是一串指令一样。这些指令包括加载多个寄存器（LDM），存储多个寄存器（STM），多个寄存器参与的 PUSH，以及多个寄存器参与的 POP。
- 除非系统被彻底地锁定，NMI（不可屏蔽中断）会在收到请求的第一时间予以响应。对于很多安全-关键(safety-critical)的应用，NMI 都是必不可少的（如化学反应即将失控时的紧急停机）。

低功耗

- Cortex-M3 需要的逻辑门数少，所以先天就适合低功耗要求的应用（功率低于 0.19mW/MHz）
- 在内核水平上支持节能模式（SLEEPING 和 SLEEPDEEP 位）。通过使用“等待中断指令（WFI）”和“等待事件指令（WFE）”，内核可以进入睡眠模式，并且以不同的方式唤醒。另外，模块的时钟是尽可能地分开供应的，所以在睡眠时可以把 CM3 的大多数“官能团”给停掉。
- CM3 的设计是全静态的、同步的、可综合的。任何低功耗的或是标准的半导体工艺均可放心饮用。

系统特性

- 系统支持“位寻址带”操作（8051 位寻址机制的“威力大幅加强版”），字节不变的大端模式，并且支持非对齐的数据访问。
- 拥有先进的 fault 处理机制，支持多种类型的异常和 faults，使故障诊断更容易。
- 通过引入 banked 堆栈指针机制，把系统程序使用的堆栈和用户程序使用的堆栈划清界线。如果再配上可选的 MPU，处理器就能彻底满足对软件健壮性和可靠性有严格要求的应用。

调试支持

- 在支持传统的 JTAG 基础上，还支持更新更好的串行线调试接口。
- 基于 CoreSight 调试解决方案，使得处理器哪怕是在运行时，也能访问处理器状态和存储器内容。
- 内建了对多达 6 个断点和 4 个数据观察点的支持。

- 可以选配一个 ETM，用于指令跟踪。数据的跟踪可以使用 DWT
- 在调试方面还加入了以下的新特性，包括 fault 状态寄存器，新的 fault 异常，以及闪存修补（patch）操作，使得调试大幅简化。
- 可选 ITM 模块，测试代码可以通过它输出调试信息，而且“拎包即可入住”般地方便使用。

第3章

Cortex-M3基础

- 寄存器组
- 特殊功能寄存器组
- 操作模式
- 异常和中断
- 向量表
- 存储器保护单元
- 堆栈区的操作
- 复位序列

寄存器组

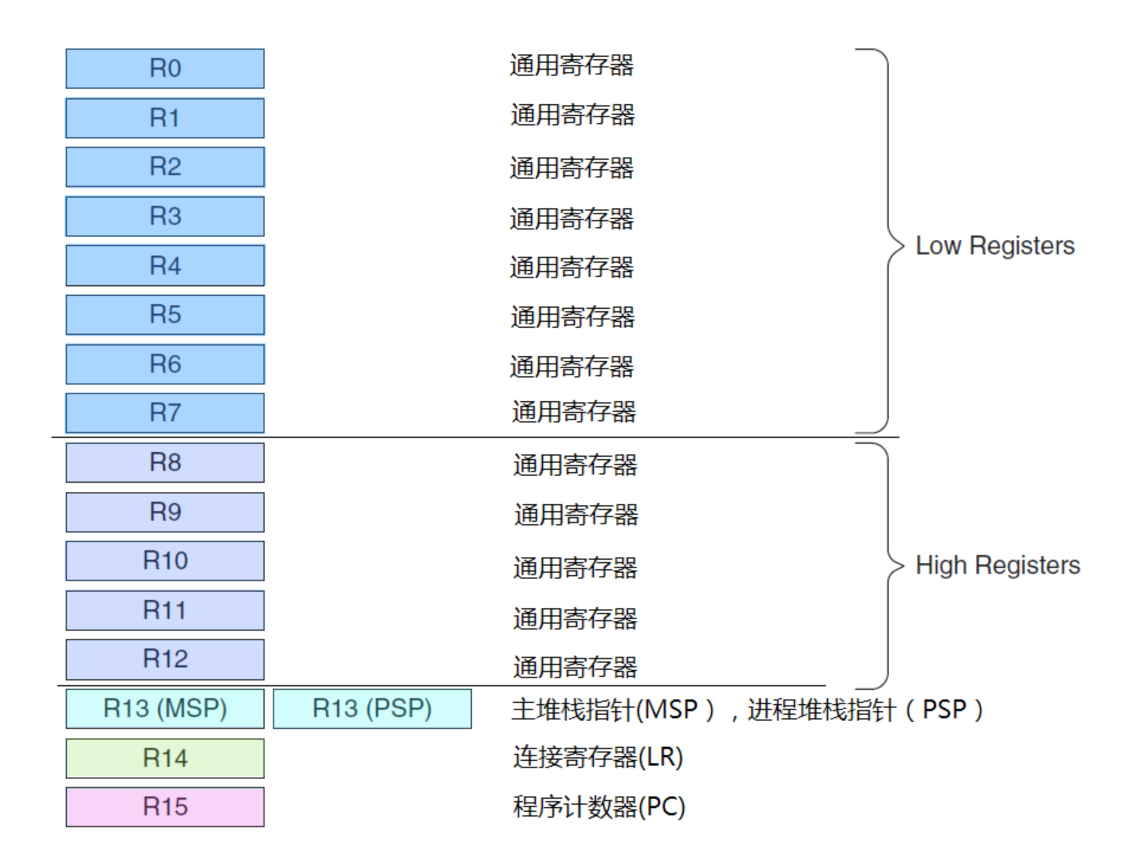
如我们所见，CM3 拥有通用寄存器 R0-R15 以及一些特殊功能寄存器。R0-R12 是最“通用目的”的，但是绝大多数的 16 位指令只能使用 R0-R7（低组寄存器），而 32 位的 Thumb-2 指令则可以访问所有通用寄存器。特殊功能寄存器有预定义的功能，而且必须通过专用的指令来访问。

通用目的寄存器 R0-R7

R0-R7 也被称为低组寄存器。所有指令都能访问它们。它们的字长全是 32 位，复位后的初始值是不可预料的。

通用目的寄存器 R8-R12

R8-R12 也被称为高组寄存器。这是因为只有很少的 16 位 Thumb 指令能访问它们，32 位的指令则不受限制。它们也是 32 位字长，且复位后的初始值是不可预料的。



特殊功能寄存器：

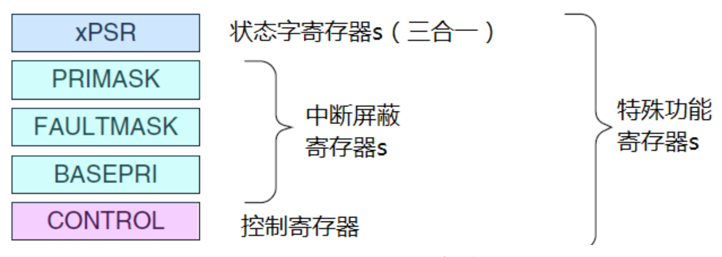


图 3.1　Cortex-M3 的寄存器组

堆栈指针 R13

R13 是堆栈指针。在 CM3 处理器内核中共有两个堆栈指针，于是也就支持两个堆栈。当引用 R13（或写作 SP）时，你引用到的是当前正在使用的那一个，另一个必须用特殊的指令来访问（MRS,MSR 指令）。这两个堆栈指针分别是：

- 主堆栈指针（MSP），或写作 SP_main。这是缺省的堆栈指针，它由 OS 内核、异常服务例程以及所有需要特权访问的应用程序代码来使用。
- 进程堆栈指针（PSP），或写作 SP_process。用于常规的应用程序代码（不处于异常服用例程中时）。

要注意的是，并不是每个应用都必须用齐两个堆栈指针。简单的应用程序只使用 MSP 就够了。堆栈指针用于访问堆栈，并且 PUSH 指令和 POP 指令默认使用 SP。

堆栈的 PUSH 与 POP

堆栈是一种存储器的使用模型。它由一块连续的内存，以及一个栈顶指针组成，用于实现“先进后出”的缓冲区。其最典型的应用，就是在数据处理前先保存寄存器的值，再在处理任务完成后从中恢复先前保护的这些值。

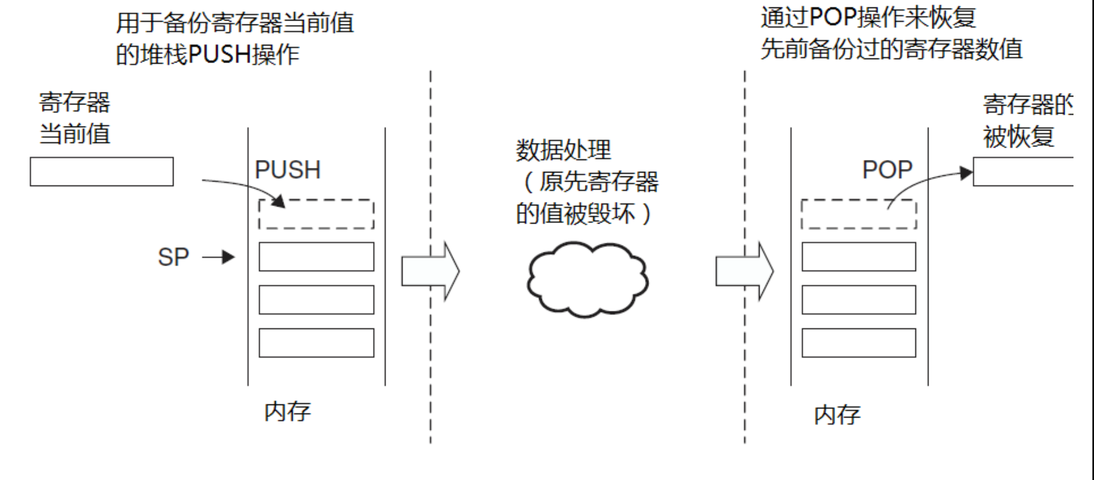


图 1.2　堆栈内存的基本概念

在执行 PUSH 和 POP 操作时，那个通常被称为 SP 的地址寄存器，会自动被调整，以避免后续的操作破坏先前的数据。本书的后续章节还要围绕着堆栈展开更详细的论述。

在 Cortex-M3 中，有专门的指令负责堆栈操作——PUSH 和 POP。它俩的汇编语言语法如下例所演示

```
PUSH    {R0}        ; *(--R13)=R0。R13 是 long*的指针
POP     {R0}        ; R0= *R13++
```

请注意后面 C 程序风格的注释，ortex-M3 中的堆栈以这种方式来使用的，这就是所谓的“向下生长的满栈”（本章后面在讲到堆栈内存操作时还要展开论述）。因此，在 PUSH 新数据时，堆栈指针先减一个单元。通常在进入一个子程序后，第一件事就是把寄存器的值先 PUSH 入堆栈中，在子程序退出前再 POP 曾经 PUSH 的那些寄存器。另外，PUSH 和 POP 还能一次操作多个寄存器，如下所示：

```
subroutine_1
    PUSH    {R0-R7, R12, R14}       ; 保存寄存器列表
    …                               ; 执行处理
    POP {R0-R7, R12, R14}           ; 恢复寄存器列表
    BX  R14                         ; 返回到主调函数
```

在程序中为了突出重点，你可以使用 SP 表示 R13。在程序代码中，both MSP 和 PSP 都被称为 R13/SP。不过，我们可以通过 MRS/MSR 指令来指名道姓地访问具体的堆栈指针。

MSP，亦写作 SP_main，这是复位后缺省使用堆栈指针，服务于操作系统内核和异常服务例程；而 PSP，亦写作 SP_process，典型地用于普通的用户线程中。

寄存器的 PUSH 和 POP 操作永远都是 4 字节对齐的——也就是说他们的地址必须是 0x4,0x8,0xc,……。这样一来，R13 的最低两位被硬线连接到 0,并且总是读出 0（Read As Zero）。

连接寄存器 R14

R14 是连接寄存器（LR）。在一个汇编程序中，你可以把它写作 both LR 和 R14。LR 用于在调用子程序时存储返回地址。例如，当你在使用 BL(分支并连接，Branch and Link)指令时，就自动填充 LR 的值。

```
main        ;主程序
    …
    BL  function1      ; 使用“分支并连接”指令呼叫 function1
                    ;  PC= function1，并且 LR=main 的下一条指令地址
    …

Function1
    …               ; function1 的代码
    BX  LR          ; 函数返回（如果 function1 要使用 LR，必须在使用前 PUSH，
                    ; 否则返回时程序就可能跑飞了——译注）
```

尽管 PC 的 LSB 总是 0（因为代码至少是字对齐的），LR 的 LSB 却是可读可写的。这是历史遗留的产物。在以前，由位 0 来指示 ARM/Thumb 状态。因为其它有些 ARM 处理器支持 ARM 和 Thumb 状态并存，为了方便汇编程序移植，CM3 需要允许 LSB 可读可写。

程序计数器 R15

R15 是程序计数器，在汇编代码中你也可以使用名字“PC”来访问它。因为 CM3 内部使用了指令流水线，读 PC 时返回的值是当前指令的地址+4。比如说：

```
0x1000:        MOV    R0,    PC     ;  R0 = 0x1004
```

如果向 PC 中写数据，就会引起一次程序的分支（但是不更新 LR 寄存器）。CM3 中的指令至少是半字对齐的，所以 PC 的 LSB 总是读回 0。然而，在分支时，无论是直接写 PC 的值还是使用分支指令，都必须保证加载到 PC 的数值是奇数（即 LSB=1），用以表明这是在 Thumb 状态下执行。倘若写了 0，则视为企图转入 ARM 模式，CM3 将产生一个 fault 异常。

特殊功能寄存器组

Cortex-M3 中的特殊功能寄存器包括：

- 程序状态寄存器组（PSRs 或曰 xPSR）
- 中断屏蔽寄存器组（PRIMASK, FAULTMASK,以及 BASEPRI）
- 控制寄存器（CONTROL）

它们只能被专用的 MSR 和 MRS 指令访问，而且它们也没有存储器地址。

```
MRS     <gp_reg>,       <special_reg>    ;读特殊功能寄存器的值到通用寄存器
MSR     <special_reg>,  <gp_reg>         ;写通用寄存器的值到特殊功能寄存器
```

程序状态寄存器（PSRs 或曰 PSR）

程序状态寄存器在其内部又被分为三个子状态寄存器：

- 应用程序 PSR（APSR）
- 中断号 PSR（IPSR）
- 执行 PSR（EPSR）

通过 MRS/MSR 指令，这 3 个 PSRs 即可以单独访问，也可以组合访问（2 个组合，3 个组合都可以）。当使用三合一的方式访问时，应使用名字“xPSR”或者“PSR”。

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

图 3.3 Cortex-M3 中的程序状态寄存器（xPSR）

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception Number				

图 3.4 合体后的程序状态寄存器(xPSR)

PRIMASK, FAULTMASK 和 BASEPRI

这三个寄存器用于控制异常的使能和除能。

表 3.2 Cortex-M3 的屏蔽寄存器 s

名字	功能描述
PRIMASK	这是个只有 1 个位的寄存器。当它置 1 时，就关掉所有可屏蔽的异常，只剩下 NMI 和硬 fault 可以响应。它的缺省值是 0，表示没有关中断。
FAULTMASK	这是个只有 1 个位的寄存器。当它置 1 时，只有 NMI 才能响应，所有其它的异常，包括中断和 fault，通通闭嘴。它的缺省值也是 0，表示没有关异常。
BASEPRI	这个寄存器最多有 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断，0 也是缺省值。

对于时间-关键任务而言，PRIMASK 和 BASEPRI 对于暂时关闭中断是非常重要的。而 FAULTMASK 则可以被 OS 用于暂时关闭 fault 处理机能，这种处理在某个任务崩溃时可能需要。因为在任务崩溃时，常常伴随着一大堆 faults。在系统料理“后事”时，通常不再需要响应这些 fault——人死帐清。总之 FAULTMASK 就是专门留给 OS 用的。

要访问 PRIMASK, FAULTMASK 以及 BASEPRI，同样要使用 MRS/MSR 指令,如：

```
MRS     R0,         BASEPRI         ;读取 BASEPRI 到 R0 中
MRS     R0,         FAULTMASK       ;似上
```

```
MRS     R0,         PRIMASK        ;似上
MSR     BASEPRI,    R0             ;写入 R0 到 BASEPRI 中
MSR     FAULTMASK, R0              ;似上
MSR     PRIMASK,    R0             ;似上
```

只有在特权级下，才允许访问这 3 个寄存器。

其实，为了快速地开关中断，CM3 还专门设置了一条 CPS 指令，有 4 种用法

```
CPSID   I       ;PRIMASK=1,         ;关中断
CPSIE   I       ;PRIMASK=0,         ;开中断
CPSID   F       ;FAULTMASK=1,       ;关异常
CPSIE   F       ;FAULTMASK=0        ;开异常
```

控制寄存器（CONTROL）

控制寄存器用于定义特权级别，还用于选择当前使用哪个堆栈指针。

表 3.3 Cortex-M3 的 CONTROL 寄存器

位	功能
CONTROL[1]	堆栈指针选择 0=选择主堆栈指针 MSP（复位后缺省值） 1=选择进程堆栈指针 PSP 在线程或基础级（没有在响应异常——译注），可以使用 PSP。在 handler 模式下，只允许使用 MSP，所以此时不得往该位写 1。
CONTROL[0]	0=特权级的线程模式 1=用户级的线程模式 Handler 模式永远都是特权级的。

CONTROL[1]

在 Cortex-M3 的 handler 模式中，CONTROL[1]总是 0。在线程模式中则可以为 0 或 1。

仅当处于特权级的线程模式下，此位才可写，其它场合下禁止写此位。改变处理器的模式也有其它的方式：在异常返回时，通过修改 LR 的位 2，也能实现模式切换。这将在第 5 章中展开论述。

CONTROL[0]

仅当在特权级下操作时才允许写该位。一旦进入了用户级，唯一返回特权级的途径，就是触发一个（软）中断，再由服务例程改写该位。

CONTROL 寄存器也是通过 MRS 和 MSR 指令来操作的：

```
MRS     R0,         CONTROL
MSR     CONTROL,    R0
```

操作模式

Cortex-M3 支持 2 个模式和两个特权等级。

	特权级	用户级
异常handler的代码	handler模式	错误的用法
主应用程序的代码	线程模式	线程模式

图 3.6 操作模式和特权等级

当处理器处在线程状态下时，既可以使用特权级，也可以使用用户级；另一方面，handler 模式总是特权级的。在复位后，处理器进入线程模式＋特权级。

在线程模式＋用户级下，对系统控制空间（SCS）的访问将被阻止——该空间包含了配置寄存器 s 以及调试组件的寄存器 s。除此之外，还禁止使用 MSR 访问刚才讲到的特殊功能寄存器——除了 APSR 有例外。谁若是以身试法，则将 fault 伺候。

在特权级下的代码可以通过置位 CONTROL[0]来进入用户级。而不管是任何原因产生了任何异常，处理器都将以特权级来运行其服务例程，异常返回后将回到产生异常之前的特权级。用户级下的代码不能再试图修改 CONTROL[0]来回到特权级。它必须通过一个异常 handler，由那个异常 handler 来修改 CONTROL[0]，才能在返回到线程模式后拿到特权级。

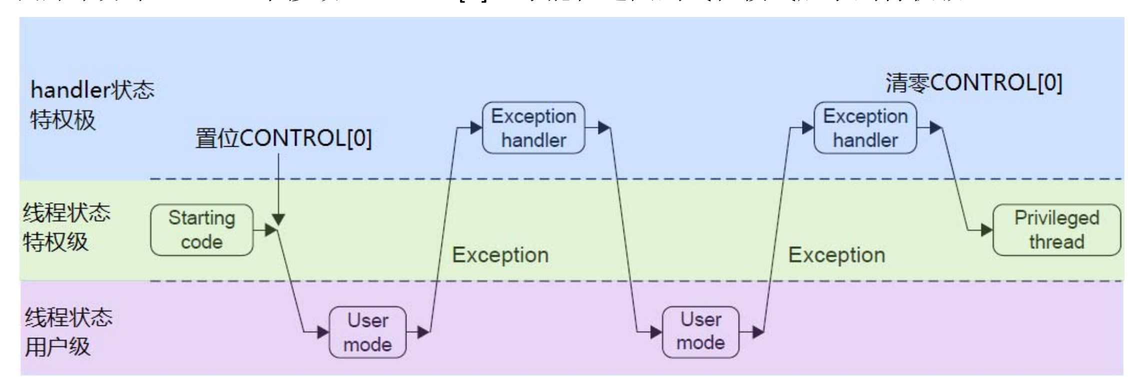


图 3.7 特权级和处理器模式的改变图

把代码按特权级和用户极分开对待，有利于使架构更加安全和健壮。例如，当某个用户代码出问题时，不会让它成为害群之马，因为用户级的代码是禁止写特殊功能寄存器和 NVIC 中寄存器的。另外，如果还配有 MPU，保护力度就更大，甚至可以阻止用户代码访问不属于它的内存区域。

为了避免系统堆栈因应用程序的错误使用而毁坏，你可以给应用程序专门配一个堆栈，不让它共享操作系统内核的堆栈。在这个管理制度下，运行在线程模式的用户代码使用 PSP，而异常服务例程则使用 MSP。这两个堆栈指针的切换是全自动的，就在出入异常服务例程时由硬件处理。第 8 章将详细讨论此主题。

如前所述，特权等级和堆栈指针的选择均由 CONTROL 负责。当 CONTROL[0]=0 时，在异常处理的始末，只发生了处理器模式的转换，如下图所示。

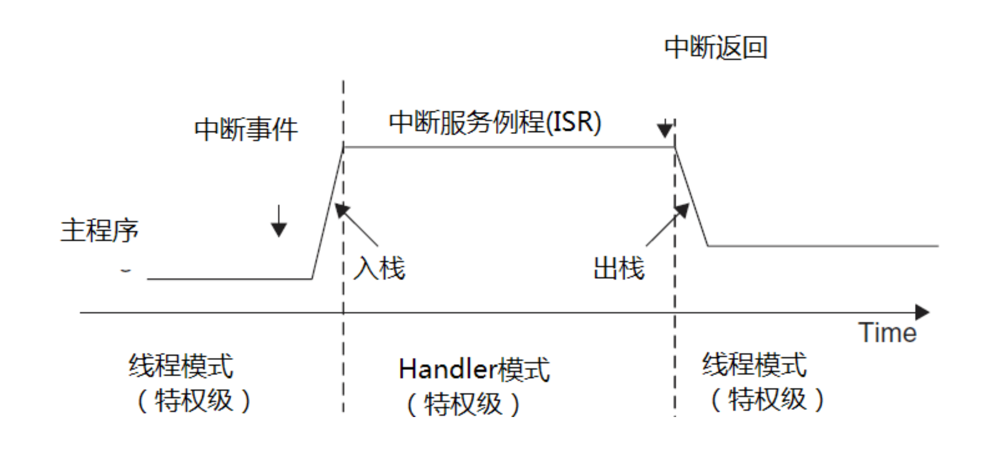


图 3.8 中断前后的状态转换

但若 CONTROL[0]=1（线程模式+用户级），则在中断响应的始末，both 处理器模式和特权等极都要发生变化，如下图所示。

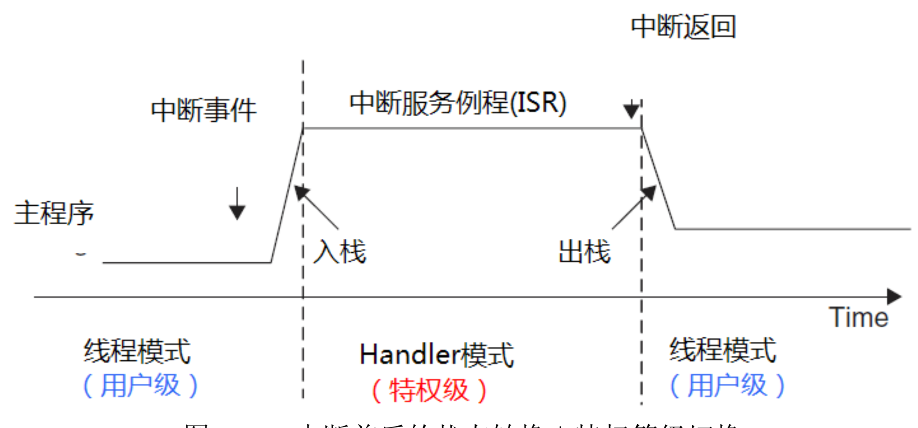


图 3.9 中断前后的状态转换＋特权等级切换

CONTROL［0］只有在特权级下才能访问。用户级的程序如想进入特权级，通常都是使用一条“系统服务呼叫指令（SVC）”来触发“SVC 异常”，该异常的服务例程可以选择修改 CONTROL[0]。

异常与中断

Cortex-M3 支持大量异常，包括 16-4-1=11 个系统异常，和最多 240 个外部中断——简称 IRQ。具体使用了这 240 个中断源中的多少个，则由芯片制造商决定。由外设产生的中断信号，除了 SysTick 的之外，全都连接到 NVIC 的中断输入信号线。典型情况下，处理器一般支持 16 到 32 个中断，当然也有在此之外的。

作为中断功能的强化，NVIC 还有一条 NMI 输入信号线。NMI 究竟被拿去做什么，还要视处理器的设计而定。在多数情况下，NMI 会被连接到一个看门狗定时器，有时也会是电压监视功能块，以便在电压掉至危险级别后警告处理器。NMI 可以在任何时间被激活，甚至是在处理器刚刚复位之后。

表 3.4 列出了 Cortex-M3 可以支持的所有异常。有一定数量的系统异常是用于 fault 处理的，它们可以由多种错误条件引发。NVIC 还提供了一些 fault 状态寄存器，以便于 fault 服务例程找出导致异常的具体原因。

表 3.4 Cortex-M3 中的异常类型

编号	类型	优先级	简介
0	N/A	N/A	没有异常在运行
1	复位	-3（最高）	复位
2	NMI	-2	不可屏蔽中断（来自外部 NMI 输入脚）
3	硬(hard) fault	-1	所有被除能的 fault，都将"上访"成硬 fault。除能的原因包括当前被禁用，或者 FAULTMASK 被置位。
4	MemManage fault	可编程	存储器管理 fault，MPU 访问犯规以及访问非法位置均可引发。企图在"非执行区"取指也会引发此 fault
5	总线 fault	可编程	从总线系统收到了错误响应，原因可以是预取流产（Abort）或数据流产，或者企图访问协处理器
6	用法(usage) Fault	可编程	由于程序错误导致的异常。通常是使用了一条无效指令，或者是非法的状态转换，例如尝试切换到 ARM 状态
7-10	保留	N/A	N/A
11	SVCall	可编程	执行系统服务调用指令（SVC）引发的异常
12	调试监视器	可编程	调试监视器（断点，数据观察点，或者是外部调试请求
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的"可悬挂请求"（pendable request）
15	SysTick	可编程	系统滴答定时器（也就是周期性溢出的时基定时器——译注）
16	IRQ #0	可编程	外中断#0
17	IRQ #1	可编程	外中断#1
...	...	...	...
255	IRQ #239	可编程	外中断#239

第 7-9 章给出了异常操作的详细信息。

向量表 s

当一个发生的异常被 CM3 内核接受，对应的异常 handler 就会执行。为了决定 handler 的入口地址，CM3 使用了"向量表查表机制"。这里使用一张向量表。向量表其实是一个 WORD（32 位整数）数组，每个下标对应一种异常，该下标元素的值则是该异常 handler 的入口地址。向量表的存储位置是可以设置的，通过 NVIC 中的一个重定位寄存器来指出向量表的地址。在复位后，该寄存器的值为 0。因此，在地址 0 处必须包含一张向量表，用于初始时的异常分配。

表 3.5 向量表结构

异常类型	表项地址偏移量	异常向量
0	0x00	MSP 的初始值
1	0x04	复位
2	0x08	NMI
3	0x0C	硬 fault
4	0x10	MemManage fault
5	0x14	总线 fault

6	0x18	用法 fault
7-10	0x1c-0x28	保留
11	0x2c	SVC
12	0x30	调试监视器
13	0x34	保留
14	0x38	PendSV
15	0x3c	SysTick
16	0x40	IRQ #0
17	0x44	IRQ #1
18-255	0x48-0x3FF	IRQ #2 - #239

举个例子，如果发生了异常 11（SVC），则 NVIC 会计算出偏移移量是 11x4=0x2C，然后从那里取出服务例程的入口地址并跳入。0 号异常的功能则是个另类，它并不是什么入口地址，而是给出了复位后 MSP 的初值。

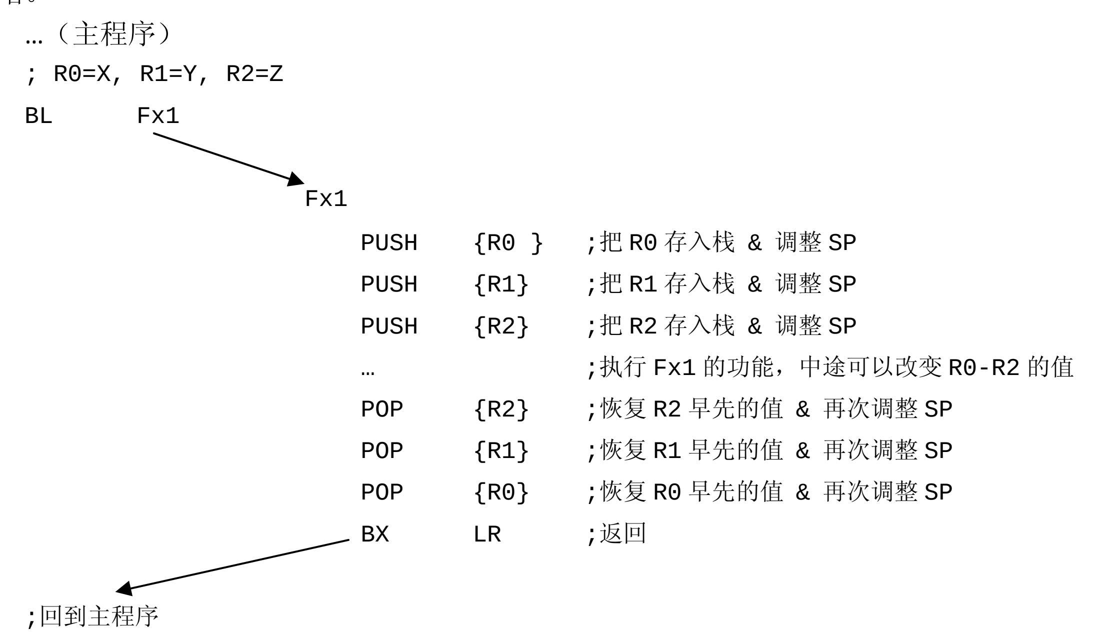
栈内存操作

在 Cortex-M3 中，除了可以使用 PUSH 和 POP 指令来处理堆栈外，内核还会在异常处理的始末自动地执行 PUSH 与 POP 操作。本节让我们来检视一下具体的动作，第 9 章则讨论异常处理时的自动栈操作。

堆栈的基本操作

笼统地讲，堆栈操作就是对内存的读写操作，但是其地址由 SP 给出。寄存器的数据通过 PUSH 操作存入堆栈，以后用 POP 操作从堆栈中取回。在 PUSH 与 POP 的操作中，SP 的值会按堆栈的使用法则自动调整，以保证后续的 PUSH 不会破坏先前 PUSH 进去的内容。

堆栈的功能就是把寄存器的数据放入内存，以便将来能恢复之——当一个任务或一段子程序执行完毕后恢复。正常情况下，PUSH 与 POP 必须成对使用，而且参与的寄存器，不论是身份还是先后顺序都必须完全一致。当 PUSH/POP 指令执行时，SP 指针的值也根着自减/自增。



;R0=X, R1=Y, R2=Z （调用 Fx1 的前后 R0-R2 的值没有被改变）

图 3.10 基本的堆栈操作：每次处理单个寄存器

译者添加：

如果参与的寄存器比较多，这种 PUSH 和 POP 岂不是又臭又长？放心，PUSH/POP 指令足够体贴，支持一次操作多个寄存器。像这样：

```
PUSH    {R0-R2}          ;压入 R0-R2
PUSH    {R3-R5,R8, R12}  ;压入 R3-R5,R8，以及 R12
```

在 POP 时，可以如下操作：

```
POP     {R0-R2}          ;弹出 R0-R2
POP     {R3-R5,R8, R12}  ;弹出 R3-R5，R8，以及 R12
```

注意：不管在寄存器列表中，寄存器的序号是以什么顺序给出的，汇编器都将把它们升序排序。然后 PUSH 指令按照从大到小的顺序依次入栈，POP 则按从小到大的顺序依次出栈。如果不按升序写寄存器，有些汇编器可能会给出一个语法错误。

PUSH/POP 对子还有这样一种特殊形式，形如

```
PUSH    {R0-R3, LR}
POP     {R0-R3, PC}
```

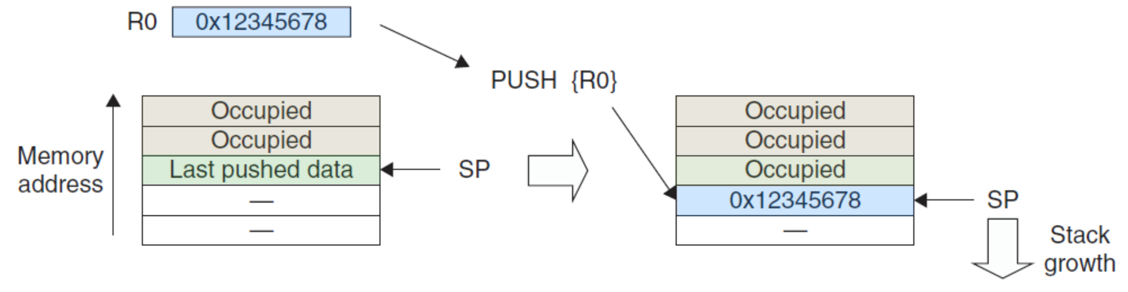
请注意：POP 的最后一个寄存器是 PC，并不是先前 PUSH 的 LR。这其实是一个返回的小技巧。因为总要把先前 LR 的值弹出来，再使用此值返回，干脆绕过 LR，直接传给 PC！那不怕 LR 的值没有被恢复吗？不怕，因为 LR 在子程序返回时的唯一用处就是提供返回地址，在返回后，先前保存的返回地址就没有利用价值了，所以只要 PC 得到了正确的值，不恢复也没关系。

PUSH 指令等效于与使用 R13 作为地址指针的 STMDB 指令，而 POP 指令则等效于使用 R13 作为地址指针的 LDMIA 指令——STMDB/LDMIA 还可以使用其它寄存器作为地址指针。至于这两个指令的细节，后续章节讲到指令系统时再介绍。

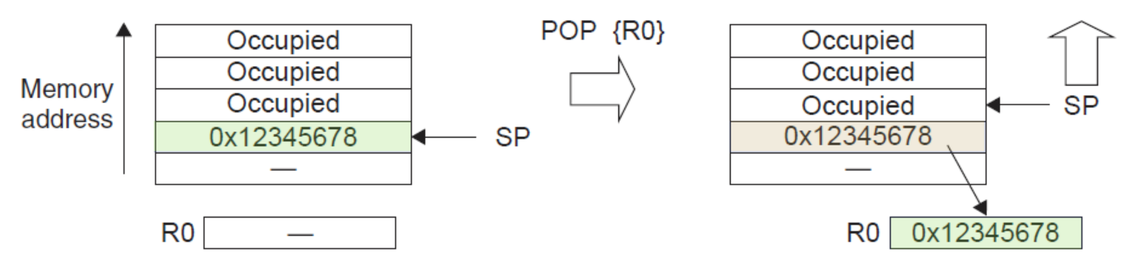
图 3.10 中的子程序返回后，R0-R2 的值仍然是执行前的——仿佛什么事都没有发生一样。

Cortex-M3 堆栈的实现

Cortex-M3 使用的是“向下生长的满栈”模型。堆栈指针 SP 指向最后一个被压入堆栈的 32 位数值。在下一次压栈时，SP 先自减 4，再存入新的数值。



POP 操作刚好相反：先从 SP 指针处读出上一次被压入的值，再把 SP 指针自增 4。



译注[9]：虽然 POP 后被压入的数值还保存在栈中，但它已经无效了，因为为下次的 PUSH 将覆盖它的值！

在进入 ISR 时，CM3 会自动把一些寄存器压栈，这里使用的是进入 ISR 之前使用的 SP 指针（MSP 或者是 PSP）。离开 ISR 后，只要 ISR 没有更改过 CONTROL[1]，就依然使用先前的 SP 指针来执行出栈操作。

再论 Cortex-M3 的双堆栈机制

我们已经知道了 CM3 的堆栈是分为两个：主堆栈和进程堆栈，CONTROL[1]决定如何选择。

当 CONTROL[1]=0 时，只使用 MSP，此时用户程序和异常 handler 共享同一个堆栈。这也是复位后的缺省使用方式。

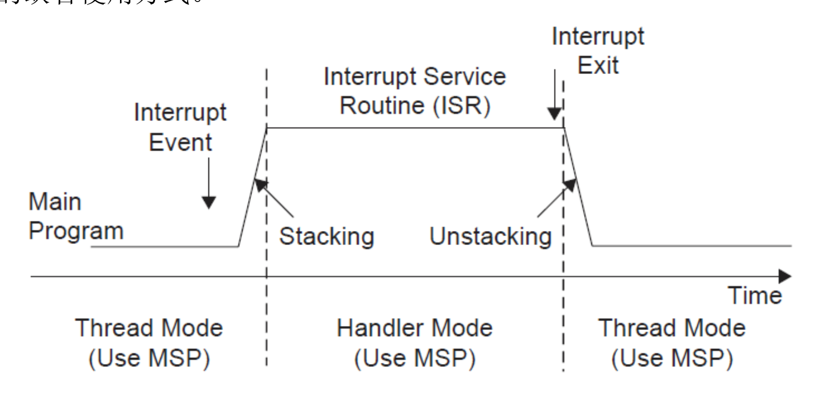


图 3.15 CONTROL[1]=0 时的堆栈使用情况

当 CONTROL[1]=1 时，线程模式将不再使用 PSP，而改用 MSP（handler 模式永远使用 MSP）。

［译注 10］：此时，进入异常时的自动压栈使用的是进程堆栈，进入异常 handler 后才自动改为 MSP，退出异常时切换回 PSP，并且从进程堆栈上弹出数据。

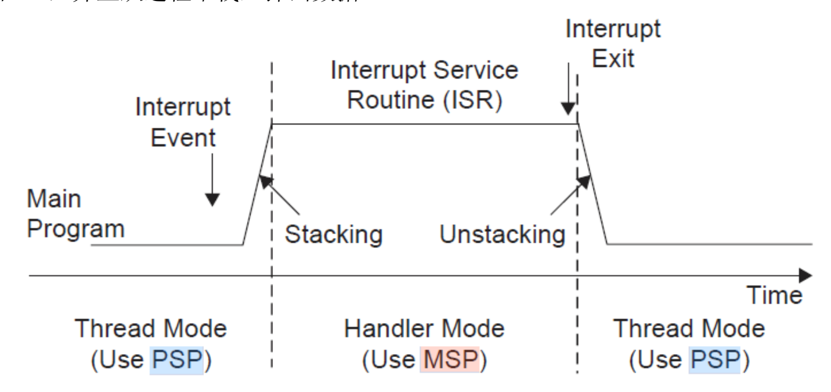


图 3.16 CONTROL[1]=0 时的堆栈切换情况

在特权级下，可以指定具体的堆栈指针，而不受当前使用堆栈的限制，示例代码如下：

```
MRS     R0,     MSP     ; 读取主堆栈指针到 R0
MSR     MSP,    R0      ; 写入 R0 的值到主堆栈中
MRS     R0,     PSP     ; 读取进程堆栈指针到 R0
MSR     PSP,    R0      ; 写入 R0 的值到进程堆栈中
```

通过读取 PSP 的值，OS 就能够获取用户应用程序使用的堆栈，进一步地就知道了在发生异常时，被压入寄存器的内容，而且还可以把其它寄存器进一步压栈（使用 STMDB 和 LDMIA 的书写形式）。OS 还可以修改 PSP，用于实现多任务中的任务上下文切换。

复位序列

在离开复位状态后，CM3 做的第一件事就是读取下列两个 32 位整数的值：

- 从地址 0x0000,0000 处取出 MSP 的初始值。
- 从地址 0x0000,0004 处取出 PC 的初始值——这个值是复位向量，LSB 必须是 1。然后从这个值所对应的地址处取指。
-

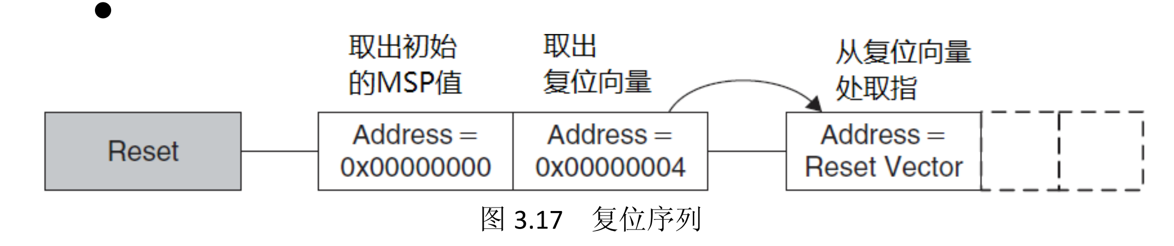


图 3.17 复位序列

请注意，这与传统的 ARM 架构不同——其实也和绝大多数的其它单片机不同。传统的 ARM 架构总是从 0 地址开始执行第一条指令。它们的 0 地址处总是一条跳转指令。在 CM3 中，0 地址处提供 MSP 的初始值，然后就是向量表（向量表在以后还可以被移至其它位置）。向量表中的数值是 32 位的地址，而不是跳转指令。向量表的第一个条目指向复位后应执行的第一条指令。

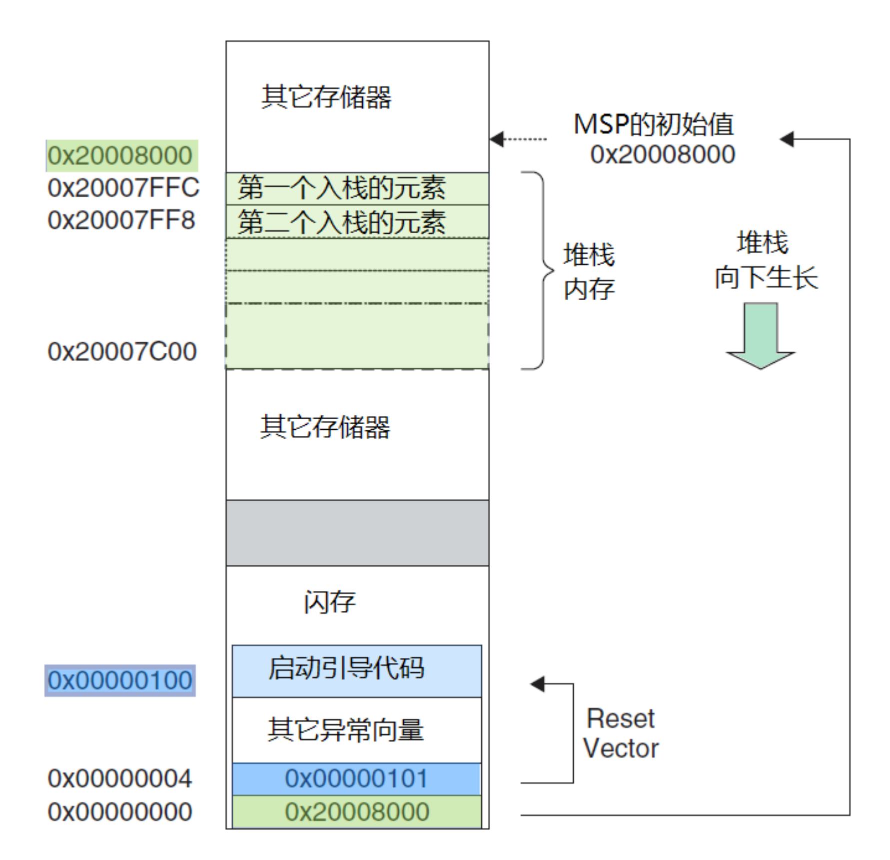


图 3.18　初始 MSP 及 PC 初始化的一个范例

因为 CM3 使用的是向下生长的满栈，所以 MSP 的初始值必须是**堆栈内存的末地址加 1**。举例来说，如果你的堆栈区域在 0x20007C00-0x20007FFF 之间，那么 MSP 的初始值就必须是 0x20008000。

向量表跟随在 MSP 的初始值之后——也就是第 2 个表目。要注意因为 CM3 是在 Thumb 态下执行，所以向量表中的每个数值都必须把 LSB 置 1（也就是奇数）。正是因为这个原因，图 3.18 中使用 0x101 来表达地址 0x100。当 0x100 处的指令得到执行后，就正式开始了程序的执行。在此之前初始化 MSP 是必需的，因为可能第 1 条指令还没执行就会被 NMI 或是其它 fault 打断。MSP 初始化好后就已经为它们的服务例程准备好了堆栈。

对于不同的开发工具，需要使用不同的格式来设置 MSP 初值和复位向量——有些则由开发工具自行计算。如果想要获知细节，最快的办法就是参考开发工具提供的一个示例工程。本书的第 10 章和第 20 章介绍 ARM 提供的开发工具，第 19 章则介绍 GCC 工具链。